AN EFFICIENT PARALLEL MAX-FLOW ALGORITHM

by

Yossi Shiloach* and Uzi Vishkin**

Technical Report #196,
February 1981

* IBM Scientific Center, Technion City, Haifa, Israel

** Computer Science Dept., Technion-Israel Institute of Technology, Haifa, Israel.
AN EFFICIENT PARALLEL MAX-FLOW ALGORITHM

Yossi Shiloach
IBM Israel Scientific Center
Haifa, Israel

Uzi Vishkin*
Computer Science Department
Technion-Israel Institute of Technology
Haifa, Israel

ABSTRACT

A synchronized parallel algorithm for finding maximum flow in a directed flow network is presented. Its depth is $O(n^3 \log n) / p$ where $p (p \leq n)$ is the number of processors used. This problem seems to be much more involved than most of the problems for which efficient parallel algorithms exist.

Our algorithm induces a new simple and interesting sequential $O(n^3)$ algorithm. This algorithm is very much parallel oriented and could hardly be found and analyzed by one who is restricted to the sequential point of view. This proves that the discipline of parallel computation can enrich the sequential way of thinking.

* This paper constitutes a part of the author's research towards his D.Sc. degree.
1. BASICS

A DIRECTED FLOW NETWORK $N = (G,s,t,c)$ is a quadruple, where:

(i) $G = (V,E)$ is a directed graph;
(ii) $s$ and $t$ are distinct vertices, the source and the terminal respectively;
(iii) $c: E \to \mathbb{R}^+$ assigns a non-negative capacity $c(e)$ to each $e \in E$.

A directed flow network is a 0-1 NETWORK if $c(e) = 1$ for all $e \in E$.

Let $u \to v$ denote a directed edge from $u$ to $v$. $d_{\text{in}}(v)$ ($d_{\text{out}}(v)$) denotes the number of edges entering (emanating from) $v$ in $G$.

A function $f: E \to \mathbb{R}^+$ is a FLOW if it satisfies:

(a) The CAPACITY rule:

$$f(e) \leq c(e) \text{ for all } e \in E.$$  

(b) The CONSERVATION rule:

$$\text{IN}(f,v) = \text{OUT}(f,v) \text{ for all } v \in V - \{s,t\}.$$  

Here $\text{IN}(f,v) = \sum_{u \to v \in E} f(u \to v)$ is the total flow entering $v$, and $\text{OUT}(f,v) = \sum_{v \to u \in E} f(v \to u)$ is the total flow emanating from $v$.

The flow VALUE $|f|$ is $\text{OUT}(f,s) - \text{IN}(f,s)$.

A flow $f$ is a MAXIMUM FLOW if $|f| \geq |f'|$ for any other $f'$.

A flow $f$ SATURATES an edge $e$ if $f(e) = c(e)$.

A flow $f$ is MAXIMAL if every directed path from $s$ to $t$ contains at least one saturated edge.

A directed network $N = (G,s,t,c)$ is called a LAYERED NETWORK if $G$ has the following properties:
(i) Each vertex \( v \) has a layer number \( \ell(v) \).

(ii) \( \ell(s) = 0 \) and \( 0 < \ell(v) < \ell(t) \) for all \( v \in V \).

(iii) If \( u \rightarrow v \in E \) then \( \ell(v) - \ell(u) = 1 \)

The set \( L_j = \{v: \ell(v) = j\} \) is called the \( j \)-th LAYER of \( G \).

2. INTRODUCTION

This paper presents a synchronized parallel algorithm for finding maximum flow in a directed network which is very simple conceptually.

Our algorithm is not only simple as a parallel algorithm. It also induces a new \( O(n^3) \) sequential algorithm that is strikingly simple. Despite its simplicity, it is not a 'natural' sequential algorithm and would hardly be found by one who looks at the problem from the sequential point of view. Its complexity proof is even more parallel-oriented and very hard to conceive through the sequential approach. This proves that the discipline of parallel computation can enrich the sequential way of thinking.

The model used is a synchronized parallel computation model in which all the processors have access to a common memory. Simultaneous reading from the same memory location is allowed. Simultaneous writing in the same memory location is also allowed provided that all the processors attempt to write the same. In fact, the simultaneous writing is not crucial and the same performance can be achieved without it. This assumption is introduced because it simplifies the implementation of the algorithm. For a detailed description of this model and its basic definitions, see [SV-80a] or [Vi-81].

The depth of the algorithm is \( O(n^3 \log n/p) \) where \( n = |V| \) and
p \leq n \) is the number of processors used. The same algorithm when applied to 0-1 networks has depth of \( O((nm \log n)/p) \) for \( p \leq m/n \) processors where \( m = |E| \), (see the second comment at the end of Sec. 6).

There are two main difficulties in designing a good parallel algorithm for the max-flow problem.

First, it seems to be more involved than most of the problems for which good parallel algorithms exist in the literature. So far, there exist good parallel algorithms for problems like finding the maximum, merging and sorting and for elementary graph problems like computing connected components, finding minimum spanning tree, performing a breadth first search on a graph and so on. Solutions to problems of this kind can be found in [AE-80], [B-68], [EA-80], [Ec-77], [Ev-74], [Ga-75], [HCS-79], [Hi-78], [P-78], [RC-78], [Sa-77], [SV-80a], [SV-80b], and [Va-75]. Another class of efficient parallel algorithms exists for problems in the field of numerical algebra; see the survey paper [He-78]. These problems also seem to have a simpler structure than the max-flow problem.

The second difficulty lies in the apparent sequential nature of this problem. Its well-known good one-processor algorithms do not have a straightforward parallel implementation. These algorithms include [GN-80], [K-74], [MPM-78] and [Sh-79].

The next section contains a high-level description of the algorithm and its validity proof. The fourth section describes the sequential algorithm and analyzes its complexity. A detailed parallel implementation of the algorithm and the data-structure involved are given in Sec. 5. A more efficient parallel implementation is given in Sec. 6. Section 7 contains the proof of the 2n bound on the number of facts which is
crucial for the evaluation of the complexity of the sequential implementation in Sec. 4 and the depth of the parallel implementation in Sec. 6. ('Tacts' are defined in the next section.)

3. HIGH-LEVEL DESCRIPTION OF THE ALGORITHM AND ITS VALIDITY PROOF.

The algorithm follows the scheme related to E.A. Dinic [D170] of transforming one maximum flow problem in a general network into $O(n)$ maximal flow problems in layered networks. We shall first discuss the problem of finding maximal flow in a given layered network which is the main problem. The transition from one layered network to the other will be briefly discussed afterwards.

3.1 Finding Maximal Flow in a Layered Network

The algorithm is divided into 'tacts'. In the first tact the source saturates all the edges emanating from it. In the beginning of each of the succeeding tacts there will be a set of BALANCED vertices (for which $IN(f,v) = OUT(f,v)$) and a set of UNBALANCED vertices satisfying $IN(f,v) > OUT(f,v)$. The balanced vertices remain idle during the tact while the unbalanced vertices try to push forward as much of the excess flow as possible. If they cannot get rid of all the excess flow this way, they return the rest backwards. Returning the flow backwards is done in a 'last in first out' (LIFO) order.

DéFINITION: A vertex $v$ for which $l(v) = l(t) - 1$ becomes BLOCKED as soon as the edge $v + t$ becomes saturated. If $l(v) < l(t) - 1$ then it becomes BLOCKED as soon as all its emanating edges are either saturated or lead to already blocked vertices.
In order to present the algorithm in a little bit more detailed ALGOL-like form, we first describe two routines. One for pushing the excess flow from a vertex forward and the other for returning the excess flow from a blocked vertex. These two routines make use of the following terms.

EXCESS(v) (=IN(f,v) - OUT(f,v)) - Denotes the excess amount of flow that should be pushed forward or returned backward from a given vertex v.

AVAILABLE(v) - Denotes the set of edges emanating from v that are neither saturated nor lead to a blocked vertex.

A FLOW QUANTUM(e,q) is an amount q of flow that was pushed through an edge e at a certain time.

In order to keep the LIFO rule while returning flow from a vertex v, we keep a stack of flow quantum entering v, called STACK(v). Each flow quantum in STACK(v) has the form (e = u→v, q).

PUSH (v,EXCESS(v)):

WHILE EXCESS(v) ≠ 0 and AVAILABLE(v) ≠ ∅

DO e(=v+w) = first edge of AVAILABLE(v).

q = min {c(e)-f(e), EXCESS(v)}.

Add: Q = (e,q) to STACK(w).

f(e) = f(e) + q; EXCESS(v) = EXCESS(v) - q; EXCESS(w) = EXCESS(w) + q.

IF f(e) = c(e)

THEN delete e from AVAILABLE(v).

OD

IF AVAILABLE(v) = ∅

THEN block v and for all u such that u→v ∈ E, delete u→v from AVAILABLE(u).
RETURN \((v, \text{EXCESS}(v))\):

WHILE \(\text{EXCESS}(v) > 0\)

DO \(Q = (e = u \to v, q)\) first flow quantum in \(\text{STACK}(v)\).

\[q' = \min\{q, \text{EXCESS}(v)\}\]

\[f(e) = f(e) + q'; \ 	ext{EXCESS}(v) = \text{EXCESS}(v) - q';\]

\[\text{EXCESS}(u) = \text{EXCESS}(u) + q'.\]

IF \(q = q'\)

THEN delete \(Q\) from \(\text{STACK}(v)\).

ELSE \(Q = (e, q - q').\)

OD.

MAXFLOW (Scheme):

TACT 1: Saturate all the edges emanating from \(s\) and block it;

\[\text{EXCESS}(v) = c(s \to v) \text{ for all } v \in V; i + 1.\]

WHILE there exist unbalanced vertices

DO \(i = i + 1\)

TACT \(i\) at \(v\): IF \(v\) is unbalanced

THEN IF \(v\) is not blocked

THEN \text{PUSH}(v, \text{EXCESS}(v)).

IF \(\text{EXCESS}(v) > 0\)

THEN \text{RETURN}(v, \text{EXCESS}(v)).

ELSE \text{RETURN}(v, \text{EXCESS}(v)).

OD

Remark: The sequential description of PUSH and RETURN can be quite misleading. It is shown in Sections 5 and 6 that in fact they have very efficient parallel implementations.
3.2 Preparation of a Layered Network

After finding a maximal flow in one layered network, another one should be constructed. This is done in two stages. First, a new (not necessarily layered) network is constructed. This network contains an edge \( e = u \rightarrow v \) iff one of the following conditions holds:

(i) \( f(e) < c(e) \). In this case the new capacity is \( c(e) - f(e) \).
(ii) \( f(v \rightarrow u) > 0 \). The new capacity here is \( f(v \rightarrow u) \).

In the second stage, Breadth-First-Search (BFS) is applied on the network above yielding a new layered network whose underlying graph is a subgraph of the new network's underlying graph. The parallel BFS algorithm also consists of tactics. In the first tact, a search from \( s \) is performed and the first layer is found. In the \( i \)-th tact, a search from the \((i-1)\)-th layer is executed revealing the \( i \)-th layer. The detailed parallel implementation of this algorithm includes several quite simple techniques. These techniques are described and used in the implementation of the maximal flow algorithm too. The way in which they are implemented in the BFS algorithm will be clear after their introduction in Section 5 and 6. A parallel implementation of the BFS algorithm can also be found in [AE-80].

The following lemma exhibits an interesting property of the algorithm.

**Lemma 3.2.1** If \( i \) is even (odd) then all the unbalanced vertices at the beginning of tact \( i \) lie in odd (even) layers.

**Proof:** Immediate by induction on \( i \). \( \square \)
3.3 Validity. Proof

In order to prove the validity of the algorithm, it should only be shown that it yields a maximal flow in each layered network. Thus, the algorithm, in this subsection refers just to that of finding maximal flow in a layered network.

Theorem 3.3.1

(a) The algorithm terminates after at most \((n+1)\ell\) tacts, \((\ell\) is the number of layers).

(b) When the algorithm terminates, it yields maximal flow.

Proof: (a) This assertion follows immediately from the fact that if no vertex is blocked during \(\ell\) successive tacts then all the flow reaches \(t\) and the algorithm terminates. This rough estimate on the number of tacts will be greatly improved in Section 7.

(b) When the algorithm terminates, all the vertices are balanced.

Let \(P = [u_0 \rightarrow u_1 \rightarrow \ldots \rightarrow u_\ell], (u_0 = s, u_\ell = t)\), be an arbitrary directed path from \(s\) to \(t\) in the layered network. We shall show that \(P\) contains at least one saturated edge. Let \(u_k, (0 \leq k < \ell)\), be the closest blocked vertex to \(t\) on \(P\). Let us show that the edge \(e = u_k \rightarrow u_{k+1}\) is saturated. At the tactic in which \(u_k\) was blocked, \(e\) must have been saturated since \(u_{k+1}\) was not blocked. This edge remained saturated since \(u_{k+1}\) has never been blocked and thus never returned any flow backward. \(\square\)
4. THE SEQUENTIAL ALGORITHM

Being an implementation of the parallel algorithm, the sequential algorithm also follows Dinic's scheme. Thus the following discussion is restricted to the problem of finding maximal flow in a layered network.

Let us first describe the sequential algorithm as a one-processor implementation of the parallel scheme. During each tact our single processor handles all the vertices that were unbalanced at the beginning of this tact, one at a time. The order in which the unbalanced vertices are handled in a given tact is unimportant and any queueing of them will do. The only significant order is that among unbalanced vertices of different tacts. It turns out, however, that one queue, QUEUE, can serve the entire sequential algorithm as follows:

STEP 1: Saturate all the edges emanating from s, block it and insert to QUEUE all the unbalanced vertices.

STEP 2: WHILE QUEUE is not empty

   DO v + first vertex in QUEUE.

   IF v is not blocked

   THEN PUSH (v,EXCESS(v)).

   IF EXCESS(v) > 0

   THEN RETURN (v,EXCESS(v)).

   ELSE RETURN (v,EXCESS(v))

   Insert the newly unbalanced vertices to QUEUE.

   OD.
COMPLEXITY OF THE SEQUENTIAL ALGORITHM

Since the sequential algorithm simulates the parallel one, one tact after another, the term 'tact' still makes sense.

The elementary operations in this algorithm will be associated with either edges or pairs of the form (vertex, tact) in such a way that each edge or pair will be associated with no more than a constant number of operations. It will be shown in Section 7 that the number of tacts does not exceed 2n. This implies that the number of (vertex, tact) pairs is bounded by 2n^2 resulting a complexity of O(n^3) for the whole algorithm.

Whenever PUSH (v, EXCESS(v)) is applied and a certain edge is saturated, this edge is charged for the resulting constant number of elementary operations. For any pair (v, Tact i) there is at most one edge emanating from v through which flow was pushed in Tact i without saturating it. In this case, the pair (v, Tact i) is charged. It was shown so far that the number of operations involved in pushing does not exceed O(n^2) in each layered network. The operations involved in returning flow will now be charged on the account of either (vertex, tact) pairs or elementary pushing operations in the following way.

Whenever flow is returned from v, it cancels some flow quantum in STACK(v) that were formerly pushed to v and causes a change in at most one more flow quantum. In the first case, the appropriate pushing operation is charged and in the second, the pair (v, current tact) is charged.

The charging rules above yield an O(n^2) bound on the number of elementary returning operations in one layered network implying an O(n^3) bound on the complexity of the whole algorithm.
5. PARALLEL IMPLEMENTATION

5.1 Data-Structure

The following data-structure is the backbone of the implementation.

Given \( k \) numbers \( a_1, \ldots, a_k \) we associate with them a complete binary tree \( T(a_1, \ldots, a_k) \) so called the PARTIAL SUMS TREE, or PS-tree in short. \( T(a_1, \ldots, a_k) \) contains \( \lceil \log_2 k \rceil \) leaves. The leftmost \( k \) leaves, so called ACTIVE LEAVES, are associated with \( a_1, \ldots, a_k \) and the other leaves are associated with zeros. Each internal node \( x \) is the root of a complete subtree \( T_\chi \). It is associated with the sum of \( a_1, a_1 + 1, \ldots, a_{\lceil \log_2 k \rceil} \) which are the numbers attached to the leaves of \( T_\chi \).

An example of a PS-tree \( T(5, 2, 4, 7, 1, 6, 3) \) is shown in Figure 5.1.

![Figure 5.1](image)

Four different PS-trees will be attached to each vertex \( v \):

1. \( T\text{-OUT}(v) \): This tree has \( d_{\text{out}}(v) \) active leaves. Each such leaf is associated with one edge emanating from \( v \). The value attached to the leaf is the amount of flow that can still be pushed through its corresponding edge.

2. \( T\text{-IN}(v) \): This tree has \( n \times d_{\text{in}}(v) \) active leaves. It simulates
STACK(v). The flow quantum is recorded in its leaves from left to right in the same way in which they should have been recorded in the stack.

3. T-ACCESS (v): This tree has \(d_{in}(v)\) active leaves. Each such leaf is associated with one edge entering \(v\). It coordinates the activity of the processors that attempt to update \(STACK(v)\) simultaneously.

4. T-SUM (v): This tree has \(d_{out}(v)\) active leaves. Each such leaf is associated with an edge emanating from \(v\). It sums the amount of flow that is returned to \(v\) at a given tact.

The fifth PS-tree will be attached to each edge \(e\).

5. T-EDGE (e): This tree has \(2n\) (=number of tacts) active leaves. Each such leaf is associated with one tact. It sums the amount of flow that is returned on \(e\) at a given tact.

OPERATIONS ON PS-TREES:

Let \(T\) be a PS-tree. Every node in \(T\) will be represented in the form \(T[h, i]\) where \(h\) is its height in \(T\) and \(i\) is its serial number among the other nodes of the same height, (see Fig. 5.2). The notation \(h(T)\) stands for \(T\)'s height, (where \(h(leaf)=1\)).

![Figure 5.2](image-url)
In order to simplify the description of the operations below, it is assumed that each leaf \( T[1,1] \) of \( T \) has a processor \( P_i \) assigned to it.

**PRIMITIVE OPERATIONS**

1. **CLEAR(1):**
   
   **STEP 1:** \( j \leftarrow 1 \).
   
   **STEP 2:** \( \text{WHILE } j \leq h(T) \)
   
   \( \text{DO } T[j, \lceil j/2 \rceil] \leftarrow 0. \)
   
   \( j \leftarrow j + 1. \)
   
   **OD.**

   In this operation, \( P_i \) zeros the values of the nodes on the path from \( T[1,1] \) to the root. CLEAR can be executed simultaneously by several processors acting on the same tree. In such a case, simultaneous writing of 0 in the same location may occur.

2. **UPDATE(1, a):**
   
   **STEP 1:** \( T[1,1] \leftarrow a. \)
   
   **STEP 2:** \( j \leftarrow 2. \)

   **WHILE \( j \leq h(T) \)**

   \( \text{DO } T[j, \lceil j/2 \rceil] \leftarrow \)

   \( T[j-1, \lceil j/2 \rceil - 1] + T[j-1, 2 \lceil j/2 \rceil]. \)

   \( j \leftarrow j + 1. \)
   
   **OD.**

   In this operation, the value of the \( i \)-th leaf is set to \( a \), and the resulting changes in the values of other nodes are performed. Several such changes can be performed at the same time.
3. SUM(i;Si):

STEP 1: \( S_i + a_i; i + 2. \)

STEP 2: WHILE \( i < h(T) \)

DO IF \( 2[i/2(j-1)] \leq [i/2(j-2)] \)

THEN \( S_i + S_i + T[j-1], [i/2(j-2)] - 1. \)

\( j + j+1 \)

OD.

SUM(i) performs: \( S_i + a_1 + \ldots + a_i. \)

4. FIND(a; k, \( p \)):

STEP I: \( j + h(T); k + 1; p + a \)

STEP 2: WHILE \( j > 1 \)

DO IF \( p > T[i-1, 2k-1] \)

THEN \( p = p - T[i-1, 2k-1]; \ k + 2k. \)

ELSE \( k + 2k-1. \)

\( j + j-1 \)

OD.

Given \( a, \) FIND returns \( k \) and \( p \) satisfying:

\( a_1 + \ldots + a_{k-1} < a < a_1 + \ldots + a_k \) and \( p = a - (a_1 + \ldots + a_{k-1}). \)

5.2 The Parallel Implementation.

It is enough to describe the implementation of PUSH and RETURN.

Having done this, the rest of MAX-FLOW's implementation can be easily accomplished by following its scheme above. It should be noted though that the implementation below requires that the RETURN operations in each tact will not start before the end of the PUSH operations of the same tact.
In order to simplify the description below, it is assumed that each vertex \( v \) (edge \( e \)) has a processor \( P(v), (P(e)) \), attached to it. Moreover, every leaf of every tree \( T-IN(v) \) has a processor assigned to it. In the course of the algorithm, flow-quantums will be associated with these leaves and the processors attached to them will be denoted as \( P(Q) \).

In the next section it is shown that a clever allocation of processors to jobs reduces the apparent big number of processors required.

The notation \( P(\cdot) \) before an instruction below denotes that it is carried out only by processors of the indicated type.

**INITIALIZATION**

In the beginning of the entire algorithm, (before the first phase), the values in the nodes of all the trees are set to 0.

The following routine is applied simultaneously for each \( v \in V \), at the beginning of each phase:

**INITIALIZE(v):**

**STEP 1:** \( P(e_j = v + w): a: UPDATE(j, c'(e)) \) in \( T-OUT(v) \).

*Here \( j \) is the serial number of \( e \) among the edges emanating from \( v \) and thus \( j \) is also the index of the leaf of \( T-OUT(v) \) associated with \( e_j \). \( c' \) is its new capacity in the current layered network.*

\( b: f(e_j) + 0. \)

*In this section \( f(e) \) denotes the flow on \( e \) restricted to current phase of the algorithm.*

**STEP 2:** \( P(v): \) \( hd(v) +0; \) \( k'(v) +1; \)

*\( hd(v) \) points to the head of \( STACK(v) \), i.e. to the rightmost
significant leaf in $T$-IN$(v)$. $k'_i(v)$ denotes the smallest index of an edge in AVAILABLE$(v)$.

Remarks:

1. In the following routines there are several variables that depend on $v$. These are $k'(v), k(v), \alpha(v)$ and $\rho(v)$. They appear as $k'$, $k$, $\alpha$ and $\rho$ for short since no ambiguity arises.

2. In the following $T$[root] denotes the value attached to the root of a tree $T$, namely $T[h(T), 1]$.

**PUSH**(v, EXCESS(v)):

STEP 1: P(v): $a + \min(\text{EXCESS}(v), T\text{-OUT}(v)[\text{root}])$.

*The value in the root of $T$-OUT$(v)$ denotes the amount of flow that can still be pushed from $v$. Thus $a$ is the amount that is pushed in the current tact.*

EXCESS$(v) \leftarrow \text{EXCESS}(v) - a$.

FIND $(a; k, \rho)$ in $T$-OUT$(v)$.

*Processor P(v) finds the edges $e_k, \ldots, e_{k'}$ through which flow is going to be pushed from $v$. The edges $e_k, \ldots, e_{k-1}$ will be saturated and an amount of $\rho$ will be pushed through $e_k$.*

STEP 2: P($e_j = v + w$): IF $k' < j < k$

THEN $a$: UPDATE($r, 1$) in $T$-ACCESS$(w)$

*The number $r$ denotes the index of the leaf of $T$-ACCESS$(w)$ that corresponds to $e_j$.*

b: SUM($r; Sr$) in $T$-ACCESS$(u)$.

*Sr is the serial number of $P(e_j)$ among the processors that wish to record flow quantums in STACK$(w)$.*
c: \( q_j + T-\text{OUT}(v)_{[1:k]} \) for \( k' < j < k \).
\( q_j' \leftarrow \rho \) if \( j = k \).

d: \( f(e_j') + f(e_j) + q_j' \).

\( *q_j', k' \leq j < k, \) is the amount of flow that is going to be pushed through \( e_j \).*

e: \( \text{TOTAL}(w) + T-\text{IN}(w)[\text{root}] \).

\( *\text{TOTAL}(w) \) is the total amount of flow that was pushed into \( w \) so far.*

f: \( \text{UPDATE}(\text{hd}(w) + Sr; q_j) \) in \( T-\text{IN}(w) \).

\( *\)The number \( \text{hd}(w) + Sr \) is the index of the leaf of \( T-\text{IN}(w) \) that corresponds to the flow quantum \( (e_j, q_j) \). This flow quantum is recorded at \( \text{STACK}(w) \) and \( T-\text{IN}(w) \) is properly updated.*

g: \( \text{UPDATE}(j, T-\text{OUT}(v)[1:j]-q_j) \)
in \( T-\text{OUT}(v) \).

\( *\)The residual capacity of \( e_j \) is updated.*

h: \( \text{hd}(w) \leftarrow \text{hd}(w) + T-\text{ACCESS}(w)[\text{root}] \).

\( *\)After \( \text{STEP 2-a} \), \( T-\text{ACCESS}(w)[\text{root}] \) contains the number of flow quantum
that were recorded in \( \text{STACK}(w) \) in the current tact. Thus the new
\( \text{hd}(w) \) points to the new head of \( \text{STACK}(w) \).*

i: \( \text{CLEAR}(r) \) in \( T-\text{ACCESS}(w) \).

\( *T-\text{ACCESS}(w) \) is cleared for possible use in succeeding tacts.*

j: \( \text{EXCESS}(w) + T-\text{IN}(w)[\text{root}] - \text{TOTAL}(w) \).

\( \text{STEP 3: } P(v); k' = k. \)

\( \text{STEP 4: } P(e_d = u + v); \text{IF } \text{EXCESS}(v) > 0 \)
\( \text{THEN } \text{BLOCK}(v) \leftarrow \text{'yes'} \)
\( \text{UPDATE}(d, 0) \) in \( T-\text{OUT}(u) \).

\( *\)The number \( d \) denotes the index of the leaf of \( T-\text{OUT}(u) \) that corresponds to the edge \( u \rightarrow v \). Since \( e_d \) leads to a blocked vertex it is
'deleted' from \( \text{AVAILABLE}(u) \).*
RETURN(v, EXCESS(v)):

STEP 1: P(v): FIND (T-IN(v)[root] - EXCESS(v); k, p) in T-IN(v).
*Returning flow from v involves the canceling of an appropriate
number of flow quantums from STACK(v). By the operation above, P(v)
finds out which quantums should still stay in the stack. The rest
(excluding one) will be deleted.*

EXCESS(v) + 0.

STEP 2: P(Q_j = (e_j = u + v, q_j));
   a: d_j + q_j for k < j < hd(v).  
   d_j + q_j - p if j = k.
   b: UPDATE (j, 0) in T-IN(v)
      for k < j < hd(v)
      UPDATE (j, p) in T-IN(v) if j = k.
*STACK(v) is properly updated.*

c: UPDATE (r_j, d_j) in T-EDGE(e_j).
*The number r_j denotes the tact at which Q_j was pushed. It is also
the index of the leaf of T-EDGE(e_j) that corresponds to this tact. In
this instruction the total amount of flow that is returned on e_j at
the current tact is figured out and stored at T-EDGE(e_j)[root].*

d: f(e_j) + f(e_j) - T-EDGE(e_j)[root].

e: UPDATE (l_j, T-EDGE(e_j)[root]
      in T-SUM(u).
*The number l_j is the index of the leaf of T-SUM(u) that corresponds
to e_j. In this instruction the total amount of flow that is currently
returned to u is found and stored at T-SUM(u)[root].*

f: EXCESS(u) + EXCESS(u) + T-SUM(u)[root].

g: CLEAR (r_j) in T-EDGE(e_j).
h: CLEAR (l_j) in T-SUM(u).
STEP 3: \( P(v); \text{hd}(v) + k \).

The following routine is applied at the end of each phase. It cleans \( T_{-\text{OUT}}(v) \) and \( T_{-\text{IN}}(v) \) for possible use in the next phase:

CLEAN\((v)\):

STEP 1: \( P(e_j = v \rightarrow w); \text{CLEAR}(j) \) in \( T_{-\text{OUT}}(v) \).

STEP 2: \( P(Q_i = (u \rightarrow v,q_i)); \text{CLEAR}(i) \) in \( T_{-\text{IN}}(v) \) for \( 1 \leq i \leq \text{hd}(v) \).

6. EFFICIENT PARALLEL IMPLEMENTATION AND ITS DEPTH ANALYSIS

The parallel implementation described in the previous section requires as much as \( n \cdot m \) processors (which is the total number of leaves in the trees \( T_{-\text{IN}}(v); v \in V \)). In this section it is shown that the number of processors can, in fact, be reduced to \( n \) without affecting the depth. It is easy to see that the depth of \( \text{PUSH} \) and \( \text{RETURN} \) is \( O(\log n) \) implying the same depth to each task. This yields a depth of \( O(n^2 \log n) \) for the whole algorithm. The framework of the more efficient implementation follows the ideas of the following theorem and its proof:

Théorème 6.1 (Brent): Any synchronized parallel algorithm of depth \( d \) that consists of \( x \) elementary operations can be implemented by \( p \) processors within a depth of \( \lceil x/p \rceil + d \).

Proof: Let \( x_i \) denote the number of operations performed by the algorithm in time \( i \), \( \sum x_i = x \). We now use the \( p \) processors to 'simulate' the algorithm. Since all the operations in time \( i \) can be executed simultaneously, they can be computed by the \( p \) processors in \( \lceil x_i/p \rceil \) units of time. Thus, the whole algorithm can be implemented by \( p \) processors in time of
Remark: The proof of Brent's theorem poses two implementation problems. The first is to evaluate $x_1$ at the beginning of time $i$ in the algorithm. The second is to assign the processors to their jobs.

In order to fully apply Brent's theorem for obtaining an algorithm of depth of $O(n^2 \log n)$ using $n$ processors, we will show the following:

1. There exists an implementation of the algorithm for which $x = O(n^3 \log n)$ and $d = \Omega(n^2 \log n)$.
2. The two problems stated above can be overcome without increasing the depth of the result.

We have already shown that the depth $d$ of the implementation above is $O(n^2 \log n)$. Let us show that the total number $x$ of elementary operations in this implementation is $O(n^3 \log n)$. At any step of PUSH or RETURN of any tact, a single processor performs at most $O(\log n)$ elementary operations. Let us define a BIG OPERATION as such a set of $O(\log n)$ elementary operations carried out by an active single processor during one step. It can be shown that the number of big operations is $O(n^3)$. The arguments are almost identical to those leading to the same bound in the analysis of the sequential algorithm in Section 4.

The following tree, T-ASSIGN, will help us to solve the allocation problems posed by Brent's Theorem. This tree has $n$ active leaves. Each such leaf is associated with a vertex. It coordinates the processors that simulate one big operation.

The number of processors that are required in order to perform a given step in PUSH or RETURN of any vertex is available before
starting the execution of this step. Steps 1 and 3 of both PUSH and
RETURN need just one processor. Step 2 of PUSH (RETURN) requires
\( k(v) - k'(v) + 1 \) processors. Step 4 of PUSH, if relevant, takes \( d_{in}(v) \) processors. It is easy to see that this number is
also available before the execution of any step of INITIALIZE and CLEAN.

The following routine uses T-ASSIGN to simulate the implementation
described in Section 5 by \( n \) processors. This routine is applied to
each step of PUSH, RETURN, INITIALIZE and CLEAN in separate.

It is assumed that both vertices and processors are numbered from
1 to \( n \). It is further assumed that \( N(j) \) is the number of processors
required by \( v_j \) in order to perform the step in hand.

SIMULATE:

STEP 1: \( P_j : \text{CLEAR}(j) \) in T-ASSIGN.
STEP 2: \( P_j : \text{UPDATE}(j,N(j)) \) in T-ASSIGN; \( k+1 \).
STEP 3: \( P_j : \text{WHILE } j+(k-1)n \leq \text{T-ASSIGN[root]}
\*x = \text{T-ASSIGN[root]} \) is the number of big operations that should be
simulated at the current step. It is done by \( \lfloor x/n \rfloor \) operations of the
loop.*

\[
\begin{align*}
\text{DO } & \quad g_1(j) = j + (k - 1)n \\
& \quad \text{FIND } (g_1(j), v(j), g_2(j)) \text{ in T-ASSIGN.} \\
& \quad \text{Perform the job of the } g_2(j)-\text{th processor of }v(j) \\
& \quad \text{at the current step.} \\
& \quad k = k + 1 \\
\text{OD}
\end{align*}
\]

Comments:

1. In order to achieve depth of \( O((n^3 \log n)/p) \) for \( p \) processors, one
just has to simulate SIMULATE with \( p \) processors.
2. MAXFLOW can be simplified when applied to 0-1 networks. The trees T-EDGE(v) are unnecessary and T-TRIN(v) should have only d_in(v) leaves.

Moreover, in the 0-1 case the complexity of the sequential algorithm is O(n^2m) as one can easily verify. Since the depth of the parallel algorithm is O(n^2 log n). The same arguments as above imply that this depth can be achieved with m/n processors. Even better results can be achieved for problems related to special 0-1 networks, (see Chap. 6 of [E79] for review of such problems). For example, maximum matching in a bipartite graph can be found within depth of O(n^3/2 log n) using m/n processors.

7. THE 2n BOUND ON THE NUMBER OF TACTS

Theorem 7.1: The algorithm terminates after at most 2n tacts.

Definition: A triple [e = u→v; ip, ir] will be called LEGAL if there was a flow quantum Q = (e, q) which was pushed through e at tact ip and was recorded then at STACK(v) and some flow returned at ir, caused a change in Q (which was at the stack's head at that time).

Note: Lemma 3.2.1 implies that if [e; ip, ir] is a legal triple then ip is odd iff ir is even.

Lemma 7.1.1: Let v_1, ..., v_k be k blocked vertices at L_j o (layer j_0). Let [e_1 = u_1 → v_1; ip_1, ir_1], ..., [e_k = u_k → v_k; ip_k, ir_k] be legal triples. Let A_j o = {a; a is even and ip_1 ≤ a ≤ ir_1 for some 1 ≤ b ≤ k}. Then the number of blocked vertices in L_j, j_o ≤ j ≤ n, upon termination is at least |A_j o|, where k = k(t).
Proof: By induction on $k - j_0$.

The induction's base is $j_0 = k - 1$. Since $Lj_0$ is the closest layer to $t$, each flow quantum that is pushed into it in a certain tact, is either returned (may be partially) in the succeeding tact or not returned at all. Hence $i_{r_b} = i_{p_b} + 1$, $1 \leq b \leq k$, and thus $|A|_j \leq k$.

However, $v_1, \ldots, v_k$ are all blocked since all of them have returned some flow. This proves the base of the induction.

The following lemma is needed for proving Lemma 7.1.1.

**Lemma 7.1.2:** Let $[e = u \rightarrow v; ip, ir]$ be a legal triple and let $ip < i < ir$. Then there exist a vertex $w$ and integers $ip_1, ir_1$ such that $[e_1 = v \rightarrow w; ip_1, ir_1]$ is a legal triple and $ip_1 \leq i \leq ir_1$.

**Proof:** Assume to the contrary that there exists $i_o, ip < i_o < ir$, for which the lemma's conditions do not hold. Let $i_b$ be the number of the tact at which $v$ was blocked, $(ip \leq i_b \leq ir)$. Let us consider two cases:

**Case 1:** $i_o \geq i_b$.

From the existence of $i_o$, such that $ip < i_o < ir$ we conclude that $ir - ip > 1$.

The fact that some amount of flow was returned from $v$ at tact $ir$, charging in $STACK(v)$ an amount of flow that was pushed into $v$ before tact $ir - 1$, implies that some flow was returned to $v$ at tact $ir - 1$.

Thus there exists a triple of the form $[v \rightarrow w; i_x, ir - 1]$. Since at tact $i_x$ some flow was pushed from $v$, $i_x < i_b$ and hence $i_x \leq i_o < ir - 1$ - a contradiction.
Case 2: \( i_0 < i_b' \).

In this case we show that any amount of flow that was pushed into \( v \) until the end of tact \( i_p \) could not be returned from \( v \). Thus, the total amount of flow that was returned from \( v \) does not exceed the amount of flow that was pushed into \( v \) after tact \( i_p \). This contradicts the existence of the triple: \([u \to v; i_p, i_r] \).

Let \( q \) be an amount of flow that was pushed on an edge entering \( v \) at a certain tact \( i_1, i_1 < i_p \). At tact \( i_1+1 \) this amount of flow was pushed on some edges emanating from \( v \). Let \( e_1 \) be such an edge. If there is no triple of the form \([e_1; i_1+1, i_2] \) then the amount of flow that was pushed through \( e_1 \) at tact \( i_1+1 \) has never been returned. If such a triple exists then \( i_2 < i_0 < i_b-1 \). Hence the amount of flow that corresponds to this triple was not returned from \( v \).

This means that there exist several edges through which this amount was pushed from \( v \) at tact \( i_2+1 \). From the same considerations as above, any amount of flow that was pushed through these edges at this tact has either not been returned at all or returned to \( v \) before tact \( i_0 \) and then rerouted.

Proof of Lemma 7.1.1 (cont'd):

Assume that the lemma holds for \( L_{j_0}^{j+1} \). Let us prove that it holds for \( L_{j_0}^{j+1} \) too. Let \( v_1, \ldots, v_k \) be \( k \) blocked vertices at \( L_{j_0}^{j+1} \) and let \([u_1 \rightarrow v_1; i_{p_1}, i_{r_1}], \ldots, [u_k \rightarrow v_k; i_{p_k}, i_{r_k}] \) be legal triples. Lemma 7.1.2 implies that there exist \( g \) blocked vertices \( w_1, \ldots, w_g \) (\( g \), integer) at \( L_{j_0}^{j+1} \) and legal triples \([v \rightarrow w_1; i_{p_1}, i_{r_1}], \ldots, [v \rightarrow w_g; i_{p_g}, i_{r_g}] \) such that \([i_{p_g} + 1, i_{r_g} - 1] \subseteq [i_{p_1}, i_{r_1}] \). (Here \([c,d] \) stands for \{\( c, c+1, \ldots, d \}\).) Thus,
\[ |A_{j_0 + 1}| + k \geq |A_{j_0}| \]  

(7.1)

Actually there might be more than one triple 'entering' a given \( w \in \{ w_1, \ldots, w_g \} \). However, if \( [v_a \rightarrow w; ip, ir] \) and \( [v_b \rightarrow w; ip', ir'] \) are legal triples, the LIFO order in \( \text{STACK}(w) \) implies that \([ip, ir]\) either contains \([ip', ir']\) or contained in it. Thus the biggest interval is chosen for each \( w \). By the induction hypothesis there are at least \( |A_{j_0 + 1}| \) blocked vertices in \( L_j \), \( j_0 + 1 \leq j < \ell \), upon termination. In addition to these blocked vertices there are \( k \) more in \( L_{j_0} \), namely \( v_1, \ldots, v_k \). By (7.1) there are at least \( |A_{j_0}| \) blocked vertices in \( L_j, j_0 \leq j < \ell \). \( \Box \)

Corollary: No flow is returned to s after tact 2n.

Proof: Assume that some flow is returned to s in tact \( t, t > 2n \), from some vertex \( u \). Thus there exists a legal triple \( [s \rightarrow u; 1, 1] \) and hence by Lemma 7.1.1 the number of blocked vertices \( \geq |A_1| > n \), - a contradiction. \( \Box \)

Proof of Theorem 7.1: Assume to the contrary that the algorithm does not terminate after 2n tacts. Thus there exists a vertex \( v \) which is unbalanced after tact 2n. Let \( E' \) be the set of all the edges that are neither saturated nor lead to a blocked vertex after tact 2n. Let us modify our layered network by assigning a new capacity \( c'(e) = f(e) \) to each \( e \in E' \). The algorithm works on the modified network in the same way as on the original one until the end of tact 2n. However, in the succeeding tacts all the excess flow in the modified network will be returned to s contradicting the corollary above. \( \Box \)
REFERENCES


REFERENCES (cont'd)


