JOURNAL OF ALGORITHMS 3, 128–146 (1982)

An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm

YOSSI SHILOACH

IBM Israel Scientific Center, Haifa, Israel

AND

UZI VISHKIN*

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

Received February 26, 1981; revised September 23, 1981

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 more involved than most of the problems for which efficient parallel algorithms exist. The parallel algorithm induces a new rather simple sequential $O(n^3)$ algorithm. This algorithm is very much parallel oriented. It is quite difficult to conceive and analyze it, if one is restricted to the sequential point of view.

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) $c: E \to R^+$ assigns a nonnegative 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 $f: E \to R^+$ is a FLOW if it satisfies:

(a) The CAPACITY rule:

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

(b) The CONSERVATION rule:

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

*This paper constitutes a part of the author's research work toward his D.Sc. degree.

0196-6774/82/020128-19$02.00/0
Copyright © 1982 by Academic Press, Inc.
All rights of reproduction in any form reserved.

Here

$$\mathrm{IN}(f, v) = \sum_{u \to v \in E} f(u \to v) \text{ is the total flow entering } v,$$

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

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

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

(ii) $l(s) = 0$ and $0 \leq l(v) \leq l(t)$ for all $v \in V$.

(iii) If $u \to v \in E$ then $l(v) - l(u) = 1$.

The set $Lj = \{v : l(v) = j\}$ is called the jth LAYER of G.

2. Introduction

This paper presents a synchronized parallel algorithm for finding maximum flow in a directed network which is quite simple conceptually. It resembles Karzanov's max-flow algorithm [13] but seems to be simpler and better adapted to parallel implementation. It also induces a new $O(n^3)$ sequential algorithm which is very simple conceptually. Despite its simplicity, it is not a "natural" sequential algorithm and has a clear parallel orientation. Both its design and time analysis are simpler if the parallel point of view is adopted.

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 value. In fact, the simultaneous writing is not crucial and it can be shown that 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 [20] or [23].

The *depth* of an algorithm is the time that passes from the moment the first processor starts operating until the last one ends. Elementary operations are assumed to take one time unit.

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 a depth of $O((nm \log n)/p)$ for $p \leq m/n$ processors, where $m = |E|$. Even better results can be achieved for problems related to special 0–1 networks. For example, maximum matching in a bipartite graph can be found within a depth of $O(n^{3/2} \log n)$ using m/n processors.

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 [1, 2, 4–6, 8, 10, 12, 15–17, 20–22]. Another class of efficient parallel algorithms exists for problems in the field of numerical algebra; see the survey paper [11]. 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 most efficient one-processor algorithms do not have a straightforward parallel implementation. These algorithms include [9, 13, 14, 18, 19].

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 are given in Section 5. The main data structure in this section is called "PS tree." Various forms of it are used in six different applications. It is proved to be useful in stack, queue, and vector maintainance and seems to be a very promising tool for other parallel algorithms.

A more efficient parallel implementation is given in Section 6. Section 7 contains the proof of the $2n$ bound on the number of pulses in the algorithm, ("pulses" are defined in the next section). This bound is crucial for the complexity evaluation of the sequential algorithm and the depth analysis of the parallel implementation in Section 6.

3. High-Level Description of the Algorithm and Its Validity Proof

The algorithm follows the scheme related to E. A. Dinic [3] of transforming one maximum flow problem in a general network into $O(n)$ maximal flow problems in layered networks. We 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 next is briefly discussed afterward.

3.1. *Finding Maximal Flow in a Layered Network*

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

Definition. A vertex v becomes BLOCKED as soon as all its emanating edges are either saturated or lead to vertices that were blocked in previous pulses. The sink t is never blocked.

In order to present the algorithm in a somewhat 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.

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

$\mathrm{AVAILABLE}(v)$: 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 pulse.

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

```
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, EXCESS(v)):
   WHILE EXCESS(v) > 0
           DO Let Q(= (e = u → v, q)) be the first flow quantum in
                   STACK(v).
                   q' ← min{q, EXCESS(v)}.
                   f(e) ← f(e) − q'; EXCESS(v) ← EXCESS(v) − q';
                   EXCESS(u) ← EXCESS(u) + q'.
                   IF q = q'
                   THEN delete Q from STACK(v).
                   ELSE Q ← (e, q − q').
           OD

   MAXFLOW(Scheme):
   PULSE 1:     EXCESS(s) ← Σ_{v∈L_1} c(s → v).
                PUSH (s, EXCESS(s))
                i ← 1.
   WHILE there exist unbalanced vertices
           DO   i ← i + 1
                PULSE i at v: IF v is unbalanced
                                   THEN IF v is not blocked
                                           THEN PUSH (v, EXCESS(v))
                                           RETURN (v, 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.

The following lemma exhibits an interesting property of the algorithm.

LEMMA 3.1.1. *If i is even (odd) then all the unbalanced vertices at the beginning of pulse i lie in odd (even) layers.*

Proof. Immediate by induction on i. □

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 $e = u \to v$ iff one of the following conditions holds:

(i) $f(e) < c(e)$. In this case the new capacity is $c(e) - f(e)$.

(ii) $f(v \to u) > 0$. The new capacity here is $f(v \to u)$.

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 pulses. In the first pulse, a search from s is performed and the first layer is found. In the ith pulse, a search from the $(i-1)$st layer is executed, revealing the ith 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 Sections 5 and 6. A parallel implementation of the BFS algorithm can also be found in [1].

3.3. *Validity Proof*

In order to prove the validity of the algorithm, it suffices to show that it yields a maximal flow in each layered network (see [3]). 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 $2l(n+1)$ pulses (l 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 $2l$ successive pulses then all the excess flow reaches the terminal vertices, s and t, and the algorithm terminates. This rough estimate on the number of pulses will be greatly improved in Section 7.

(b) When the algorithm terminates, all the vertices are balanced. Let $P = [u_0 \to u_1 \to \cdots \to u_l]$ $(u_0 = s, u_l = t)$ 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 $(0 \le k < l)$ be the closest blocked vertex to t on P. Let us show that the edge $e = u_k \to u_{k+1}$ is saturated. At the pulse 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. □

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 layered network.

Let us first describe the sequential algorithm as a one-processor implementation of the parallel scheme. During each pulse our single processor

handles all the vertices that were unbalanced at the beginning of this pulse, one at a time. The order in which the unbalanced vertices are handled in a given pulse is unimportant and any queueing of them will do. The only significant order is that among unbalanced vertices of different pulses. It turns out, however, that one queue, QUEUE, can serve the entire sequential algorithm as follows:

STEP 1: $\text{EXCESS}(s) \leftarrow \sum_{v \in L_1} c(s \to v)$

PUSH (s, EXCESS(s)) and insert to QUEUE all the unbalanced vertices.

STEP 2: WHILE QUEUE is not empty
DO
$v \leftarrow$ first vertex in QUEUE
IF v is not blocked
THEN PUSH (v, EXCESS(v)).
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 pulse after another, the term "pulse" still makes sense.

The elementary operations in this algorithm will be associated with either edges or pairs of the form (vertex, pulse) in such a way that each edge or pair will be associated with no more than a constant number of operations. It is shown in Section 7 that the number of pulses does not exceed $2n$. This implies that the number of (vertex, pulse) pairs is bounded by $2n^2$ yielding 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, Pulse i) there is at most one edge emanating from v through which flow was pushed in Pulse i without saturating it. In this case the pair (v, Pulse i) is charged. It has been 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, pulse) pairs or elementary pushing operations in the following way.

Whenever flow is returned from v, it cancels some flow quanta 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 pulse) 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 $T(a_1, \ldots, a_k)$, called the PARTIAL SUMS TREE, or PS-tree. $T(a_1, \ldots, a_k)$ contains $2^{\lceil \log_2 k \rceil}$ leaves. The leftmost k leaves, 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_x. It is associated with the sum of $a_{i_1}, a_{i_1+1}, \ldots, a_{i_2}$ which are the numbers attached to the leaves of T_x. An example of a PS-tree $T(5, 2, 4, 7, 1, 6, 3)$ is shown in Fig. 5.1.

Four different PS-trees will be attached to each vertex v.

1. T-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-IN(v): This tree has $2n \times d_{\text{in}}(v)$ active leaves ($2n$ stands for the number of pulses). It simulates STACK(v). The flow quanta are 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 $d_{\text{in}}(v)$ 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 $d_{\text{out}}(v)$ 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 pulse.

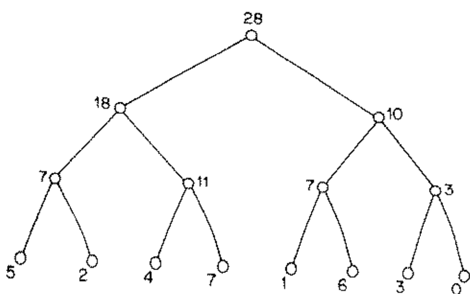

FIGURE 5.1

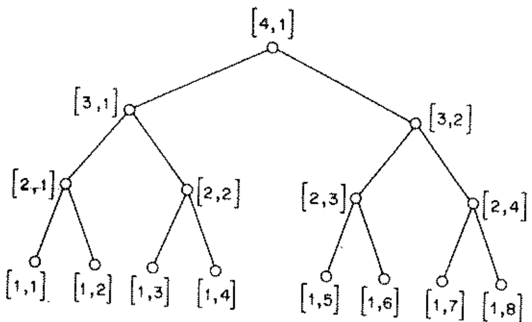

FIGURE 5.2

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

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

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 $h(\text{leaf}) = 1$).

In order to simplify the description of the operations below, it is assumed that each leaf $T[1, i]$ of T has a processor P_i assigned to it.

Primitive Operations

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

In this operation P_i zeros the values of the nodes on the path from $T[1, i]$ 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(i, a_i):
   STEP 1: $T[1, i] \leftarrow a_i$.
   STEP 2: $j \leftarrow 2$.
   WHILE $j \leq h(T)$
   DO $T[j, \lceil i/2^{(j-1)} \rceil] \leftarrow$
   $T[j-1, 2\lceil i/2^{(j-1)} \rceil - 1] + T[j-1, 2\lceil i/2^{(j-1)} \rceil]$.
   $j \leftarrow j + 1$
   OD

In this operation, the value of the ith leaf is set to a_i 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: $Si \leftarrow a_i$; $j \leftarrow 2$.
   STEP 2: WHILE $j \leq h(T)$
   DO IF $2\lceil i/2^{(j-1)} \rceil = \lceil i/2^{(j-2)} \rceil$
   THEN $Si \leftarrow Si + T[j-1, \lceil i/2^{(j-2)} \rceil - 1]$.
   $j \leftarrow j + 1$
   OD

   SUM(i) performs: $Si \leftarrow a_1 + \cdots + a_i$.

4. FIND(α; k, ρ):
   STEP 1: $j \leftarrow h(T)$; $k \leftarrow 1$; $\rho \leftarrow \alpha$.
   STEP 2: WHILE $j > 1$
   DO IF $\rho > T[j-1, 2k-1]$
   THEN $\rho \leftarrow \rho - T[j-1, 2k-1]$; $k \leftarrow 2k$
   ELSE $k \leftarrow 2k - 1$
   $j \leftarrow j - 1$
   OD

Given α, FIND returns k and ρ satisfying:

$$a_1 + \cdots + a_{k-1} < \alpha \leq a_1 + \cdots + a_k \quad \text{and} \quad \rho = \alpha - (a_1 + \cdots + a_{k-1}).$$

5.2. *The Parallel Implementation*

It is enough to describe the implementation of PUSH and RETURN. After this is done, the rest of MAX-FLOW's implementation can be accomplished easily by following its scheme above. It should be noted, however, that the implementation below requires that the RETURN operations in each pulse not start before the end of the PUSH operations of the same pulse.

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 quanta 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 careful allocation of processors to jobs reduces the apparent large number of processors required.

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

Initialization

At 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: (Comments are enclosed by asterisks.)

INITIALIZE(v):

STEP 1: $P(e_j = v \to w)$: 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. *

$$b: f(e_j) \leftarrow 0.$$

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

STEP 2: $P(v)$: $hd(v) \leftarrow 0$; $k'(v) \leftarrow 1$.

* $hd(v)$ points to the head of STACK(v), i.e., to the rightmost significant leaf in T-IN(v). $k'(v)$ 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, α, and ρ 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: $P(v)$: $\alpha \leftarrow$ Min(EXCESS(v), T-OUT(v)[root]).

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

$$\text{EXCESS}(v) \leftarrow \text{EXCESS}(v) - \alpha$$
$$\text{FIND}\ (\alpha; k, \rho) \text{ in } T\text{-OUT}(v).$$

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

STEP 2: $P(e_j = v \to w)$: IF $k' \le j \le k$

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(w).

* S_r is the serial number of $P(e_j)$ among the processors that wish to record flow quanta in STACK(w). *

c: $q_j \leftarrow T\text{-OUT}(v)[1, j]$ for $k' \le j < k$.

$q_j \leftarrow \rho$ if $j = k$.

d: $f(e_j) \leftarrow f(e_j) + q_j$.

* q_j, $k' \le j \le k$, is the amount of flow that is going to be pushed through e_j. *

e: TOTAL(w) $\leftarrow$ T-IN(w)[root].

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

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

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

g: UPDATE(j, T-OUT(v)$[1, j] - q_j$)

in T-OUT(v).

* The residual capacity of e_j is updated. *

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

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

i: CLEAR(r) in T-ACCESS(w).

* T-ACCESS(w) is cleared for possible use in succeeding pulses. *

j: EXCESS(w) ← T-IN(w)[root] − TOTAL(w).

STEP 3: $P(v)$: $k' \leftarrow k$.

STEP 4: $P(e_d = u \rightarrow v)$: IF EXCESS(v) > 0
THEN BLOCK(v) ← "yes"
UPDATE(d, 0) in T-OUT(u).

* The number d denotes the index of the leaf of T-OUT(u) that corresponds to the edge $u \rightarrow v$. Since e_d leads to a blocked vertex it is "deleted" from AVAILABLE(u). *

RETURN(v, EXCESS(v)):

STEP 1: $P(v)$: FIND (T-IN(v)[root] − EXCESS(v); k, ρ) in T-IN(v).

* Returning flow from v involves the canceling of an appropriate number of flow quanta from STACK(v). By the operation above, $P(v)$ determines which quanta should still stay in the stack. The rest (excluding one) will be deleted. *

EXCESS(v) ← 0.

STEP 2: $P(Q_j = (e_j = u \rightarrow v, q_j))$ a: $d_j \leftarrow q_j$ for $k < j \le \text{hd}(v)$.
$d_j \leftarrow q_j - \rho$ if $j = k$.
b: UPDATE(j, 0) in T-IN(v) for $k < j \le \text{hd}(v)$
UPDATE(j, ρ) 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 pulse at which Q_j was pushed. It is also the index of the leaf of T-EDGE(e_j) that corresponds to this pulse. In this instruction the total amount of flow that is returned on e_j at the current pulse is figured out and stored at T-EDGE(e_j)[root]. *

d: $f(e_j) \leftarrow f(e_j)$ − 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: $P(v)$: hd(v) ← k.

The following routine is applied at the end of each phase. It cleans T-OUT(v) and T-IN(v) for possible use in the next phase.

CLEAN(v):

STEP 1: $P(e_j = v \rightarrow w)$: CLEAR (j) in T-OUT(v).

STEP 2: $P(Q_i = (u \leftarrow v, q_i))$: CLEAR ($i$) in T-IN(v) for $1 \le i \le \text{hd}(v)$.

6. Efficient Parallel Implementation and Its Depth Analysis

The parallel implementation described in the previous section requires as much as nm processors (which is the total number of leaves in the trees T-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 PUSH and RETURN is $O(\log n)$, implying the same depth to each pulse. 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.

Theorem 6.1 (Brent). *Any synchronized parallel algorithm of depth d that consists of a total 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 $\left(\sum_1^d x_i = x \right)$. 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

$$\sum_1^d \lceil x_i/p \rceil \le \sum_1^d (x_i/p + 1) \le \lceil x/p \rceil + d. \quad \square$$

Remark. The proof of Brent's theorem poses two implementation problems. The first is to evaluate x_i 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 $d = O(n^2 \log n)$.

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 pulse, 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 $k(v) - k'(v) + 1$ $(\mathrm{hd}(v) - k(v) + 1)$ 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: CLEAR (j) in T-ASSIGN.

STEP 2: P_j: UPDATE $(j, N(j))$ in T-ASSIGN; $k \leftarrow 1$.

STEP 3: P_j: WHILE $j + (k-1)n \leq T$-ASSIGN[root]

$* x = T$-ASSIGN[root] is the number of big operations that should be simulated at the current step. It is done by $\lceil x/n \rceil$ operations of the loop. $*$

DO $g_1(j) \leftarrow j + (k-1)n$

FIND $(g_1(j); v(j), g_2(j))$ in T-ASSIGN.

Perform the job of the $g_2(j)$th processor of $v(j)$ at the current step.

$k \leftarrow k + 1$

OD

$* v(j)$ denotes the vertex to which P_j is currently assigned. $*$

Comments. (1) In order to achieve a depth of $O((n^3 \log n)/p)$ for p processors, one just has to simulate SIMULATE with p processors.

(2) MAX-FLOW can be simplified when applied to 0–1 networks. The trees T-EDGE(v) are unnecessary and T-IN(v) should have only $d_{in}(v)$ leaves. Moreover, in the 0–1 case the complexity of the sequential algorithm is $O(nm)$ as one can easily verify. Since the depth of the parallel algorithm is $O(n^2 \log n)$. The same arguments as those 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 [7, Chap. 6] for a review of such problems). For example, maximum matching in a bipartite graph can be found within a depth of $O(n^{3/2} \log n)$ using m/n processors.

7. The $2n$ Bound on the Number of Pulses

THEOREM 7.1. *The algorithm terminates after at most $2n$ pulses.*

DEFINITION. A triple $[e = u \to v;\ ip,\ ir]$ will be called LEGAL if there was a flow quantum $Q = (e, q)$ which was pushed through e at pulse 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 3.1.1 implies that if $[e;\ ip,\ ir]$ is a legal triple then ip is odd iff ir is even.

LEMMA 7.1.1. *Let $v_1, \ldots, v_k$ be k blocked vertices at Lj_0 (layer j_0). Let $[e_1 = u_1 \to v_1;\ ip_1,\ ir_1], \ldots, [e_k = u_k \to v_k;\ ip_k,\ ir_k]$ be legal triples. Let $Aj_0 = \{a\colon a$ is even and $ip_b \leq a \leq ir_b$ for some $1 \leq b \leq k\}$. Then the total number of blocked vertices in Lj, $j_0 \leq j < l$, upon termination is at least $|Aj_0|$, where $l = l(t)$.*

Proof. By induction on $l - j_0$.

The induction's base is $j_0 = l - 1$. Since Lj_0 is the layer closest to t, each flow quantum that is pushed into it in a certain pulse is either returned (perhaps partially) in the succeeding pulse or not returned at all. Hence $ir_b = ip_b + 1$, $1 \leq b \leq k$, and thus $|Aj_0| \leq k$. 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* $[e = u \to v;\ ip, ir]$ *be a legal triple and let* $ip < i < ir$. *Then there exist a vertex* w *and integers* ip_1, ir_1 *such that* $[e_1 = v \to w;\ ip_1, ir_1]$ *is a legal triple and* $ip_1 \le i \le ir_1$.

Proof. Assume to the contrary that there exists i_0, $ip < i_0 < ir$, for which the lemma's conditions do not hold. Let ib be the number of the pulse at which v was blocked ($ip \le ib \le ir$). Let us consider two cases:

Case 1. $i_0 \ge ib$.

From the existence of i_0, such that $ip < i_0 < ir$ we conclude that $ir - ip > 1$.

The fact that some amount of flow was returned from v at pulse ir, changing in STACK(v) an amount of flow that was pushed into v before pulse $ir - 1$, implies that some flow was returned to v at pulse $ir - 1$. Thus, there exists a triple of the form $[v \to w;\ i_x, ir - 1]$. Since at pulse i_x some flow was pushed from v, $i_x \le ib$ and hence $i_x \le i_0 \le ir - 1$; a contradiction.

Case 2. $i_0 < ib$.

In this case we show that any amount of flow that was pushed into v until the end of pulse ip 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 pulse ip. This contradicts the existence of the triple $[u \to v;\ ip, ir]$.

Let q be an amount of flow that was pushed on an edge entering v at a certain pulse i_1, $i_1 \le ip$. At pulse $i_1 + 1$ 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 pulse $i_1 + 1$ has never been returned. If such a triple exists then $i_2 < i_0 \le ib - 1$. 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 pulse $i_2 + 1$. From the same considerations as those above, any amount of flow that was pushed through these edges at this pulse has either not been returned at all or returned to v before pulse i_o and then rerouted. □

Proof of Lemma 7.1.1 (*continued*). Assume that the lemma holds for L_{j_0+1}. Let us prove that it holds for L_{j_0} too. Let $v_1, \ldots, v_k$ be k blocked vertices at L_{j_0} and let $[u_1 \to v_1;\ ip_1, ir_1], \ldots, [u_k \to v_k;\ ip_k, ir_k]$ 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+1} and legal triples $[v_{\alpha_1} \to w_1;\ ip'_1, ir'_1], \ldots, [v_{\alpha_g} \to w_g;\ ip'_g, ir'_g]$ such that $\bigcup_1^k [ip_a + 1, ir_a - 1] \subset \bigcup_1^g [ip'_b, ir'_b]$. (Here $[c, d]$ stands for $\{c, c+1, \ldots, d\}$.) Thus,

$$|A_{j_0+1}| + k \ge |A_{j_0}|. \tag{7.1}$$

Actually there might be more than one triple "entering" a given $w \in \{w_1, \ldots, w_g\}$. However, if $[v_\alpha \to w;\ ip, ir]$ and $[v_\beta \to w;\ ip', ir']$ are legal triples, the LIFO order in STACK(w) implies that $[ip, ir]$ either contains $[ip', ir']$ or is 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, $j_0 + 1 \le j < l$, 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, $A_{j_0} \le j < l$. □

COROLLARY. *No flow is returned to s after pulse* $2n$.

Proof. Assume that some flow is returned to s in pulse i, $i > 2n$, from some vertex u. Thus there exists a legal triple $[s \to u;\ 1, i]$ and hence by Lemma 7.1.1 the number of blocked vertices $\ge |A_1| > n$; a contradiction. □

Proof of Theorem 7.1. Assume to the contrary that the algorithm does not terminate after $2n$ pulses. Thus there exists a vertex v which is unbalanced after pulse $2n$. Let E' be the set of all the edges that are neither saturated nor lead to a blocked vertex after pulse $2n$. 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 in which it works on the original one until the end of pulse $2n$. However, in the succeeding pulses all the excess flow in the modified network will be returned to s, contradicting the corollary above. □

REFERENCES

1. D. A. ALTON AND D. M. ECKSTEIN, Parallel breadth-first search of p sparse graphs, *Utilitas Math.*, in press.
2. K. E. BATCHER, Sorting networks and their applications, *Proc. AFIPS Spring Joint Comput. Conf.* **32** (1968), 307–314.
3. E. A. DINIC, Algorithm for solution of a problem of maximum flow in a network with power estimation, *Soviet Math. Dokl.* **11** (1970), 1277–1280.
4. D. M. ECKSTEIN AND D. A. ALTON, Parallel searching of nonsparse graphs, *Siam J. Comput.*, in press.
5. D. M. ECKSTEIN, "Parallel Processing Using Depth Search and Breadth-First Search," Ph.D. thesis, Department of Computer Science, University of Iowa, Iowa City, 1977.
6. S. EVEN, Parallelism in tape-sorting, *Comm. ACM* **17**, No. 4 (1974), 202–204.
7. S. EVEN, "Graph Algorithms," Computer Science Press, Potomac, Md., 1979.
8. F. GAVRIL, Merging with parallel processors, *Comm. ACM* **18**, No. 10 (1975), 588–591.
9. Z. GALIL AND A. NA'AMAD, An $O(|E||V|\log^2|V|)$ algorithm for the maximal flow problem, *J. Comput. System Sci.* **21**, 2 (1980), 203–217.

10. D. S. HIRSCHBERG, A. K. CHANDRA, AND D. V. SARWATE, Computing connected components on parallel computers, *Comm. ACM* **22**, No. 8 (1979), 461–464.
11. D. HELLER, A survey of parallel algorithms in numerical linear algebra, *SIAM Rev.* **20**, No. 4 (1978), 740–777.
12. D. S. HIRSCHBERG, Fast parallel sorting algorithms, *Comm. ACM* **21**, No. 8 (1978), 657–661.
13. A. V. KARZANOV, Determining the maximal flow in a network by the method of preflows, *Soviet Math. Dokl.* **15** (1974), 434–437.
14. V. M. MALHORTA, M. PRAMODH KUMAR, AND S. N. MAHESHWARI, "An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks," Computer Science Program, Indian Institute of Technology, Kanpur 208016, India, 1978.
15. F. P. PREPARATA, New parallel-sorting schemes, *IEEE Trans. Comput.* **C-27** (1978), 669–673.
16. E. REGHBATI (ARJOMANDI) AND D. G. CORNEIL, Parallel computations in graph theory, *SIAM J. Comput.*, **7**, No. 2 (1978), 230–237.
17. C. SAVAGE, "Parallel Algorithms for Graph Theoretic Problems," Ph.D. thesis, University of Illinois, Urbana, 1977.
18. Y. SHILOACH, "An $O(nl \log^2 n)$ Max-Flow Algorithm," STAN-CS-78-702, Computer Science Department, Stanford University, 1978.
19. D. D. SLEATOR, "An $O(nm \log n)$ Algorithm for Maximum Network Flow," Ph.D. thesis, Stanford University, 1981.
20. Y. SHILOACH AND U. VISHKIN, Finding the maximum merging and sorting in a parallel computation model, *J. Algor.*, **2**, No. 1 (1981), 88–102.
21. Y. SHILOACH AND U. VISHKIN, An $O(\log n)$ parallel connectivity algorithm, *J. Algor.*, in press.
22. L. G. VALIANT, Parallelism in comparison problems, *SIAM J. Comput.* **4**, No. 3 (1975), 348–354.
23. U. VISHKIN, "Synchronized Parallel Computation," D.Sc. thesis, Computer Science Department, Technion, Haifa, 1981. [in Hebrew]