JOURNAL OF ALGORITHMS 13, 489–501 (1992)

A Parallel Blocking Flow Algorithm for Acyclic Networks

UZI VISHKIN*

University of Maryland, College Park, Maryland 20742 and Tel Aviv University, Tel Aviv, Israel

Received January 29, 1990; revised July 1991

Shiloach and Vishkin gave a simple parallel method for finding a blocking flow in a layered network. We show how to extend it for finding a blocking flow in any directed acyclic network and still achieve similar complexity bounds. The extended algorithm runs in $O(n \log n)$ time and $O(n^2)$ space on an n-processor EREW PRAM, where n is the number of vertices in the network. This settles an open problem raised by Goldberg and Tarjan. Goldberg–Tarjan's minimum-cost circulation algorithm together with this new–old algorithm gives an efficient parallel algorithm for the minimum-cost circulation problem on networks with integer edge costs. The algorithm runs in $O(n^2(\log n)\log(nC))$ time using n processors and $O(n^2)$ space, where C is the maximum absolute value of an edge cost. Of independent interest is the articulation of a *general* paradigm for describing parallel algorithms, called the *work time* (WT) *presentation methodology*, which has already been used as a presentation framework in JaJa's new textbook on parallel algorithms.

1. BASICS

A *directed acyclic flow network* is a quadruple $N = (G, s, t, c)$, where

(i) $G = (V, E)$ is a directed acyclic graph.

(ii) s and t are two distinct vertices, the *source* and the *sink*, respectively. No edges enter s and no edges leave t.

(iii) c: $E \to R^+$ assigns a nonnegative capacity $c(e)$ to each edge $e \in E$.

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

*Supported by NSF Grants CCR-8906949 and 9111348.

0196-6774/92 $5.00

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

(a) The CAPACITY rule: $f(e) \leq c(e)$ for each edge e in E.

(b) The CONSERVATION rule: for each vertex v in $V - \{s, t\}$, $\text{IN}(f, v)$, the sum of $f(e)$ over all edges e entering v, equals $\text{OUT}(f, v)$, the sum of $f(e)$ over all edges e emanating from v.

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

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

In the present paper we consider the problem of finding a blocking flow in an acyclic network. For comparison with [SV-82], we will need the concept of *layered networks*, where all directed paths from s to a vertex v have the same length. If the length is i then v belongs to layer i.

2. Introduction

Dinic [D-70] showed that the maximum flow problem can be solved by solving a sequence of at most n blocking flow problems on layered networks (i.e., directed acyclic flow networks where each directed path from s to t has the same number of edges). This led several papers to contribute algorithms for this problem [C-77; D-70; G-80; GN-80; K-74; MPM-78; S-78; SV-82; ST-83; T-84].

The problem of finding a blocking flow in an acyclic network has a shorter history. Goldberg and Tarjan [G-87; GT-90], showed that the minimum-cost circulation problem can be solved by solving a sequence of $O(n \log(nC))$ blocking flow problems on acyclic network, which are not necessarily layered. It is assumed that all edge costs are integers and C is the maximum absolute value of an edge cost.

Sequential algorithms for finding a blocking flow in an acyclic network. The paper [GT-89] lists the blocking flow algorithms of [D-70; GN-80; K-74; MPM-78; ST-83; T-84], as extendible to acyclic networks within similar efficiency bounds as on layered networks. The fastest sequential algorithm for the problem takes $O(m \log(n^2/m))$ time and $O(m)$ space and is described in [GT-90].

Parallel algorithms for finding a blocking flow in an acyclic network. The paper [GT-89] presents an $O(n \log n)$-time, m-processors, $O(nm)$-space EREW PRAM algorithm for the problem. The main contribution of the present paper is to use the Shiloach–Vishkin [SV-82] method to get an EREW PRAM algorithm for the problem that runs in $O(n \log n)$ time and $O(n^2)$ space using n processors.

Goldberg–Tarjan's minimum-cost circulation algorithm together with our blocking flow algorithm imply an efficient parallel algorithm for the

minimum-cost circulation problem on a network with integer edge costs bounded by some integer C. The algorithm takes $O(n^2(\log n)\log(nC))$ time and $O(n^2)$ space using n processors. We highlight its main steps. A notion of approximate optimality (called ε-optimal) of "pseudoflow" functions is defined. The algorithm works in iterations. Guided by a "scaling paradigm," each iteration aims at improving ε by a factor of two. Each iteration works by repeatedly modifying a pseudoflow until it is a circulation, while preserving the approximate optimality. For this, blocking flow in an auxiliary network is computed.

Orlin's [O-88] minimum-cost parallel circulation algorithm runs in $O(m \log^3 n)$ time using n^3 processors (by [GT-89] this can be reduced to $n^3/\log n$ processors by simply using a better parallel shortest-path algorithm). Comparison of his time results versus ours shows that his algorithm is often faster, while the time-processor product—the primary concern in evaluating parallel algorithms—in our algorithm is smaller.

The style of our presentation emphasizes the increments relative to the paper of [SV-82], which also constitutes a chapter of [V-81]. The reader is expected to have that paper in front of him/her, since we will avoid repeating identical parts. We will show how to extend the algorithm for finding a blocking flow in a layered network to finding a blocking flow in general directed acyclic networks. Sections 3–7 below correspond to Sections 3–7 in [SV-82], respectively. Section 3 overviews the algorithm and Section 4 relates to its serial implementation. These sections are essentially identical to those of [SV-82]. Section 5 overviews two aspects of the parallel implementation: (1) the work–time (WT) methodology for presenting PRAM algorithms, which was first used in [SV-82] and became a standard tool since then, is described explicitly; and (2) the 2–3 tree parallel data structure of [PVW-83] is incorporated into the parallel implementation in order to improve space complexity. Wrapping up the parallel implementation in Section 6 is again analogous to [SV-82]. The main part of the complexity analysis is given in Section 7. Section 8 concludes the paper.

3. High-Level Description of the Algorithm

The next subsection gives a high-level description of the BLOCKINGFLOW algorithm. A more specific description is borrowed from [SV-82].

3.1. *Finding a Blocking Flow in a Directed Acyclic Networks*

The algorithm works in *pulses*. In the first pulse the source s saturates all its outgoing edges. In the beginning of each pulse i, each vertex v

satisfies $\mathrm{IN}(f,v) \geq \mathrm{OUT}(f,v)$. If $\mathrm{IN}(f,v) = \mathrm{OUT}(f,v)$ then vertex v is *balanced* (in pulse i). A balanced vertex remains idle during the pulse. If $\mathrm{IN}(f,v) > \mathrm{OUT}(f,v)$ then vertex v is *unbalanced*. *All unbalanced vertices "participate" in the pulse.* Let $\mathrm{EXCESS}(v) = \mathrm{IN}(f,v) - \mathrm{OUT}(f,v)$ be the *excess flow* at vertex v. An *unbalanced* vertex v tries first to push as much excess flow as possible through its outgoing edges (this is done by a PUSH routine). If this flow pushing does not eliminate the excess flow, vertex v returns all the remaining excess flow through its incoming edges. This is done by a RETURN routine that specifies how to return flow (through incoming edges) in a "last in first out (LIFO)" order.

DEFINITIONS. A vertex v becomes *blocked* as soon as all its outgoing edges are either saturated or lead to vertices that were blocked in previous pulses. The sink t is never blocked. A blocked vertex never becomes unblocked again.

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 quanta entering v, called STACK(v). Each flow quantum in STACK(v) has the form $(e = u \to v, q)$.

We refer the reader to the statement of Algorithm MAXFLOW, in pages 131–132 of [SV-82]. It serves as the statement of Algorithm BLOCKINGFLOW of the present paper. We will use the title DIFFERENCE to itemize differences between MAXFLOW, whose inputs are layered networks, and BLOCKINGFLOW, whose inputs are directed acyclic networks.

DIFFERENCE 1. *For layered networks.* If i is even (odd) then all unbalanced vertices at the beginning of pulse i must lie in layers whose number is odd (even) (see Lemma 3.1.1 in [SV-82]). *For acyclic networks.* Consider the first pulse in which a vertex becomes unbalanced. From then on, the vertex can be unbalanced at the beginning of successive pulses.

The main contribution of this paper is a subtler analysis that copes with the more general situation. In Section 7 of this paper we prove that the number of pulses never exceeds $3n$.

DIFFERENCE 2. *For layered networks.* The number of pulses is bounded by $2n$. *For acyclic networks.* The number of pulses is bounded by $3n$.

Note that Section 3.2 of [SV-82] is irrelevant and Section 3.3 holds with the following modification: Substitute the number of layers by the length of the longest path from s to t.

4. The Sequential Implementation

The sequential implementation of the BLOCKINGFLOW algorithm and its validity proof are as in [SV-82]. The complexity argument can be summarized as follows: Claim 1. The number of flow quanta is $O(n^2 + m)$. *Proof*. We use a charging scheme to prove this. At each pulse, each vertex v that pushes flow has at most one outgoing edge that is partially saturated. Each flow quantum that saturates an edge is charged to that edge. A flow quantum that does not saturate an edge is charged to vertex v at this pulse. Since the number of pulses is $O(n)$ the claim follows. Claim 2. The sequential implementation performs $O(n^2 + m)$ (which is $O(n^2)$) operations. *Proof*. Claim 1 covers all operations where flow is pushed. Flow returned from a vertex v at a certain pulse is charged as follows. Only the last flow quantum which is popped out of the stack of vertex v may not be eliminated. If a flow quantum was eliminated then the flow return operation is charged to the flow quantum. If not then vertex v at this pulse is charged.

5. Parallel Implementation

The paper [SV-82] introduced a new methodology for describing PRAM algorithms. Since the methodology is somewhat implicit in [SV-82] (and was not given a name there), we encapsulate it below.

The WORK-TIME (WT) Methodology for Describing PRAM Algorithms. The methodology suggests describing PRAM algorithms in two stages, as follows:

(1) The algorithm is described in terms of parallel rounds (*pulses*). For each round, the operations to be performed are characterized. Several issues can be suppressed: the number of operations at each pulse need not be explicitly clear, processors need not be mentioned, in general, and assignment of processors to jobs, in particular. (We remark that intentional suppression of these issues is what makes this method useful.) We call this level of description *the work-time (WT) suppression level*, since only work (operations to be performed) and time are accounted for.

It is crucial to observe that the following theorem, due to Brent [B-74], holds for this somewhat informal WT suppression level. We remark that the target model of parallel computation used in [B-74] is similar to the WT suppression level and recall that our target model is a PRAM.

THEOREM. *A synchronous parallel algorithm that takes a total of d rounds and consists of x elementary operations can be implemented by p processors within $\lceil x/p \rceil + d$ rounds.*

Proof. Let x_i denote the number of operations performed by the algorithm at round i ($\sum_{i=1}^{d} x_i = x$). Since the p processors can "simulate" round i in $\lceil x_i/p \rceil \leq x_i/p + 1$ time units, the theorem follows. Note, however, that care must be taken if reads and writes of the same variable occur in the same step (with the assumption that all reads occur before any write).

(2) Full PRAM specification. The proof of Brent's theorem is used as a yardstick for adding full PRAM specification into parallel algorithms given for the WT suppression level. Several examples, in addition to [SV-82], (see, e.g., [J-92]) enable us to make the following circumstantial statement: *later inserting the details omitted by the WT suppression level is often not very difficult.*

Section 5 in [SV] reflects Stage 1 (i.e., description for the WT suppression level) of the WT methodology and Section 6 reflects Stage 2.

In the remainder of Section 5 and in Section 6, below, we show how to implement Algorithm BLOCKINGFLOW in $O(n \log n)$ time using $O(n^2)$ space on an n-processor EREW PRAM. In [SV-82] $\Theta(nm)$ space is needed on an n-processor CREW PRAM; the time remains $O(n \log n)$. We outline here only the changes that bring about the improvement in space with respect to [SV-82]. They are based on a result of [PVW-83], which is quoted below. Trading the CREW PRAM for the weaker EREW PRAM follows immediately from this result of [PVW-83].

THEOREM 5.1 [PVW-83]. *Consider a set of l keys from a totally ordered domain, which is represented by a 2-3 tree T with l leaves, one per key. We can implement the 2-3 tree in $O(l)$ space, to support parallel search, insert, and delete operations, as specified below. Suppose that $a_1 < a_2 < \cdots < a_k$ are keys that may or may not be stored in the leaves and there is a processor standing by each a_i, $1 \leq i \leq k$. Then, it is possible to perform each of the following three operations in $O(\log l + \log k)$ time and $O(l + k \log l)$ space on an* EREW PRAM: (1) *Search for $a_1, \cdots, a_k$ in T (that is, for each i, $1 \leq i \leq k$, determine whether a_i is in T.)* (2) *Insert $a_1, \cdots, a_k$ into T. (Following this operation, every a_i, $1 \leq i \leq k$, is a key of T.)* (3) *Delete $a_1, \cdots, a_k$ from T. (Following this operation, no a_i, $1 \leq i \leq k$, is a key of T.)*

5.1 *Data Structure*

Suppose we are given l keys $a_1 < \cdots < a_l$ and each key a_i, $1 \le i \le l$, has a number b_i associated with it. A PARTIAL SUMS 2-3-TREE (or PS-2-3-TREE) with respect to $a_1, \cdots, a_l$ is a 2-3 tree $T(a_1, \cdots, a_l)$, representing the set $\{a_1, \cdots, a_l\}$. Each internal node x is the root of a 2-3 tree $T_x(a_{i_1}, a_{i_1+1}, \cdots, a_{i_2})$ for some $1 \le i_1 \le i_2 \le l$. Node x maintains the sum of $b_{i_1}, b_{i_1+1}, \cdots, b_{i_2}$. Note that both the keys and the numbers associated with them keep changing. Theorem 5.1 will apply to all changes of keys that may occur in the parallel implementation of BLOCKINGFLOW. Maintaining sums of numbers associated with the leaves of an internal node involves some additional details, which are of trivial nature and are therefore omitted. However, the basic complexity bounds implied by Theorem 5.1 still apply.

The PS-2-3-trees replace the PS-trees of [SV-82], which are complete binary trees. In [SV-82] five PS-trees are employed. Four are attached to each vertex v (specifically, T-OUT(v), T-IN(v), T-ACCESS(v), and T-SUM(v)) and one to each edge e (T-EDGE(e)). Only two of these PS-trees cause the parallel implementation of [SV-82] to require $\Theta(nm)$ space (others need only a total of $O(n^2)$ space). These two are T-IN(v), for every vertex v, and T-EDGE(e), for every edge e. Change 1 below concerns T-IN(v) and Change 2 concerns T-EDGE(e).

Change 1. *The SV paper*. Let y denote an upper bound on the number of pulses. (Difference 2 implies that for layered networks $y = 2n$ while for acyclic ones $y = 3n$.) For each vertex v, the SV paper represents T-IN(v) by a PS-tree with $y \cdot d_{\text{in}}(v)$ active leaves. The flow quanta are recorded in its leaves from left to right in the same way they would have been recorded in a stack. Observe that the rigid structure of the PS-tree forced reserving a leaf for the worst case number of flow quanta entering node v.

The present paper. We use a single PS-2-3-tree, called T-IN, for all vertices. Initially T-IN is empty. Let $Q_1 = (u_1 \to v_1, q_1)$ be a flow quantum that was pushed into vertex v_1 in pulse t_1. For T-IN the key representing Q_1 is the triple (v_1, t_1, u_1) and the number associated with it is q_1. A lexicographic order LEX_1 is defined on the keys, as follows. Let $Q_2 = (u_2 \to v_2, q_2)$ be a flow quantum that was pushed into vertex v_2 in pulse t_2. Then, $Q_1 < Q_2$ if one of the following three holds:

(i) $v_1 < v_2$, or

(ii) $v_1 = v_2$ and $t_1 < t_2$, or

(iii) $v_1 = v_2$, $t_1 = t_2$, and $u_1 < u_2$.

In pulse t_1, flow quantum Q_1 is inserted into T-IN. Whenever flow is returned, flow quanta are either deleted (in case the returned flow

eliminated them) or the flow value is reduced (in case only part of the flow was returned). The LEX_1 order guarantees that for each vertex v, the flow quanta of STACK(v) are recorded in successive leaves of the tree in the same order as in STACK(v).

Change 2. *The SV paper.* For each edge e, the SV paper represents T-EDGE(e) by a PS-tree with y active leaves (where y is again an upper bound on the number of pulses). Each leaf is associated with one pulse. A flow quantum (e, q) in pulse t is recorded in leaf number t. Each partial (or full) return of this flow quantum causes an update of this leaf.

The present paper. We use a single PS-2-3-tree, called T-EDGE, for all edges. Initially T-EDGE is empty. Let $Q_1 = (e_1, q_1)$ be a flow quantum that was pushed on edge e_1 in pulse t_1. For T-EDGE the key representing Q_1 is the pair (e_1, t_1) and the number associated with it is q_1. A lexicographic order LEX_2 is defined on the keys as follows. Let $Q_2 = (e_2, q_2)$ be a flow quantum that was pushed on edge e_2 in pulse t_2. Then $Q_1 < Q_2$ if one of the following two holds:

(i) $e_1 < e_2$ or

(ii) $e_1 = e_2$ and $t_1 < t_2$.

In pulse t_1 flow quantum Q_1 is inserted into T-EDGE. Whenever flow is returned, flow quanta are either deleted (in case the returned flow eliminated them) or the flow value is reduced (in case only part of the flow was returned). The lexicographic order guarantees that for each edge e, its flow quanta are recorded in successive leaves of the T-EDGE tree ordered by pulse number. This enables emulating T-EDGE(e) of [SV-82].

We emphasize that for each internal node of a PS-2-3-tree we maintain the sum of the flow values of its leaves.

5.2. *The Parallel Implementation*

We refer the reader to Section 5.2 in [SV-82]. We just make a few comments here:

1. As in [SV-82], we represent the set of vertices V by $\{1, 2, \cdots, n\}$ and the set of edges E by $\{n + 1, n + 2, \cdots, n + m\}$.

2. In order to apply Theorem 5.1, keys to be searched in (or inserted into or deleted from) a PS-2-3-tree must be sorted. For this purpose, we can first place keys to be sorted into contiguous locations of an array, using a prefix sums algorithm (e.g., [LF-80]), and then apply the EREW PRAM sorting algorithm of [C-88]. It sorts n elements in $O(\log n)$ time using n processors.

3. The LEX_1 order enables us to treat the flow quanta that were pushed into each vertex v separately from other vertices, and thereby emulate T-IN(v), for each v, using T-IN. This can be done within the efficiency bounds claimed in the present paper.

4. The LEX_2 order enables us to treat the flow quanta of an edge e separately from other edges, and thereby emulate T-EDGE(e), for each edge e, using T-EDGE. This can be done within the efficiency bounds claimed in the present paper.

5. In our application, l, of Theorem 5.1, is $O(n^2)$ and $k \leq n$ and, therefore, the space requirements do not exceed $O(n^2)$.

6. The parallel implementation of Section 5 requires as many as n^2 processors (which is the total number of leaves of all PS-2-3-trees) and $O(n \log n)$ time using $O(n^2)$ space. This is improved in Section 6.

6. Efficient Parallel Implementation

No new ideas beyond those of Section 6 of [SV-82] are needed to let only n processors emulate the parallel implementation of the previous section, without affecting the bounds on time or space. Arguments similar to the analysis of the sequential implementation (specifically, Claim 1 in Section 4) upper bound the number of flow quanta by $O(n^2)$. A high-level description of the parallel implementation consists of "macro" operations (called BIG operations in [SV-82]); similar to Claim 2 in Section 4, each flow quantum corresponds to $O(1)$ macro operations; using data structures, each macro operation is executed by $O(\log n)$ PRAM operations, resulting in a total of $O(n^2 \log n)$ PRAM operations. Section 5 already explained why the time is $O(n \log n)$. We note that this whole emulation corresponds to the second stage of the WT methodology.

The main complexity result achieved in the present paper is:

Parallel Complexity Result. Algorithm BLOCKINGFLOW runs in $O(n \log n)$ time using $O(n^2)$ space on an n-processor EREW PRAM.

7. The $3n$ Bound on the Number of Pulses

The complication that makes the proof of Theorem 7.1 in [SV-82] fail for directed acyclic networks is demonstrated by the following sequence of events. Some vertex u pushes flow in some pulse i on an edge $u \to v$. Then, in pulse $i + 1$ two things happen: (1) vertex u pushes more flow on the edge $u \to v$; and (2) vertex v is blocked and returns flow on edge

$u \to v$. While such a sequence of events is possible for directed acyclic networks, it is impossible for layered networks, since for layered networks it is impossible that a vertex will push flow in two successive pulses.

THEOREM 1. *The algorithm terminates after at most* $3n$ *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 in pulse ip and was recorded then at STACK(v) while some flow returned through e at ir caused a change in Q (which was at the stack's head at that time).

Assign to each vertex v a different integer $L(v)$ between 1 and n, so that $L(t) = n$, $L(s) = 1$, and $L(u) > L(v)$ implies that there is no directed path from u to v. The $L(v)$ values provide a topological sort of the directed acyclic network.

For every vertex v ($v \neq s$), let SPAN(v) be the empty set if v never returns flow and otherwise let SPAN(v) = $\{ip, ip + 1, \cdots, ir\}$, where ip is the first pulse in which v receives flow that is (at least partially) returned later, and ir is the last pulse in which v returns flow.

LEMMA 1. *Let* v *be a vertex and* SPAN(v) = $\{i, i + 1, \cdots, j\}$. *Then there is a legal triple* $[u \to v; i, j]$.

Proof. This is implied by the LIFO rule for returning flow.

LEMMA 2. *Let* $[u \to v; ip, ir]$ *be a legal triple with* $ir - ip \geq 2$. *Then* $\{ip + 1, \cdots, ir - 1\}$ *is a subset of* $\bigcup_{v \to w \in E} \text{SPAN}(w)$.

Proof. Precisely as the proof of Lemma 7.1.2 in [SV-82].

COROLLARY. *For every*

$$v \in V - \{s\}, \ |\text{SPAN}(v)| \leq \Big| \bigcup_{v \to w \in E} \text{SPAN}(w) \Big| + 3.$$

Proof. If $|\text{SPAN}(v)| \leq 3$ there is nothing to prove and, otherwise, apply Lemma 2.

LEMMA 3. *For every integer* k *with*

$$1 \leq k \leq n - 1, \left| \bigcup_{j=n-k+1}^{n} \text{SPAN}(L^{-1}(j)) \right| \leq 3(k - 1).$$

Proof. By induction on k. The basis, $k = 1$, is trivial, since $L^{-1}(n)$ is the sink t and SPAN(t) is empty. The inductive step follows from the corollary to Lemma 2.

COROLLARY. No flow is returned to s after pulse $3n - 6$.

Proof. The source s pushes flow only in the first pulse. If vertex u returns flow to s in pulse i, then $[s \to u; 1, i]$ is a legal triple. But then, by Lemma 3, $i \leq |\text{SPAN}(u)| \leq |\bigcup_{j=2}^{n} \text{SPAN}(L^{-1}(j))| \leq 3n - 6$. Theorem 1 follows.

8. CONCLUSION

While the paper [SV-82] recognizes that its parallel method builds on the "preflow push" idea of Karzanov, it is interesting to try and trace increments in ideas and techniques that evolved out of that paper. See also the recent review in [M-89].

(i) *Conceiving the flow network itself as a distributed machine whose vertices operate synchronously in parallel.* Dinic's method solves the maximum flow problem by repeatedly deriving layered networks relative to the residual flow graph using breadth-first search (BFS) and finding blocking flows in them. We note that BFS lends itself naturally to the synchronous distributed viewpoint. Goldberg and Tarjan [GT-88] suggested the first algorithm that deviates from finding blocking flows in layered networks. Instead of finding blocking flows by one distributed algorithm and doing BFS by another, they cleverly incorporated the two into one distributed algorithm that dynamically adjusts the layered structure of the residual flow network as the blocking flow algorithm proceeds. Their new approach yielded an important new sequential algorithm.

(ii) *The LIFO order on returning flow*, which is crucial in the complexity analysis. Tarjan [T-89] implied that this idea together with (i) above and Goldberg–Tarjan's deviation from Dinic's "layered-network" scheme are some of the important recent ideas on max-flow algorithms.

(iii) *The WORK-TIME methodology for describing PRAM algorithms.* This methodology, as described in Section 5, became a standard tool. *It is called Brent's scheduling principle* in the survey paper [KR-90], and has already been adapted as the main presentation framework in [J-92]—a recent textbook on parallel algorithms.

ACKNOWLEDGMENTS

Bob Tarjan drew my attention to the problem solved in this paper. He and Andrew Goldberg read a first draft of the proof of the $3n$ bound on the number of pulses. Two

anonymous referees made numerous helpful comments that led to considerable simplifications. Omer Berkman, Baruch Schieber, and Ramakrishna Thurimella made helpful comments. Their help is gratefully acknowledged.

REFERENCES

[B-74] R. BRENT, The parallel evaluation of general arithmetic expressions, *J. Assoc. Comput. Mach.* **21** (1974), 201–206.

[C-77] R. V. CHERKASKY, Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations, *Math. Methods Solution Econom. Probl.* **7** (1977), 112–125. [Russian]

[C-88] R. COLE, Parallel merge sort, *SIAM J. Comput.* **17** (1988), 770–785.

[D-70] E. A. DINIC, Algorithm for solution of a problem of maximum flow in networks with power estimation, *Soviet Math. Dokl.* **11** (1970), 1277–1280.

[G-80] Z. GALIL, An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem, *Acta Inform.* **14** (1980), 221–242.

[GN-80] Z. GALIL AND A. NAAMAD, An $O(EV \log V)$ algorithm for the maximal flow problem, *J. Comput. System Sci.* **21** (1980), 203–217.

[G-87] A. V. GOLDBERG, "Efficient Graph Algorithms for Sequential and Parallel Computers," Ph.D. thesis, TR-374, Lab. for Computer Science, MIT, 1987.

[GT-88] A. V. GOLDBERG AND R. E. TARJAN, A new approach to the maximum flow problem, *J. Assoc. Comput. Mach.* **35** (1988), 921–940.

[GT-89] A. V. GOLDBERG AND R. E. TARJAN, A parallel algorithm for finding a blocking flow in an acyclic network, *Inform. Process. Lett.* **31** (1989), 265–271.

[GT-90] A. V. GOLDBERG AND R. E. TARJAN, Finding minimum-cost circulations by successive approximation, *Math. Oper. Res.* **15** (1990), 430–466; Conference version in "Proceedings, 19th ACM Symp. on Theory of Computing, 1987," pp. 7–18.

[J-92] J. JAJA, "An Introduction to Parallel Algorithms," Addison-Wesley, Reading, MA, 1992.

[KR-90] R. M. KARP AND V. RAMACHANDRAN, Parallel algorithms for shared memory machines, *in* "Handbook of Theoretical Computer Science. Vol. A: Algorithms and Complexity" (J. Van Leeuwen, Ed.), pp. 869–941, MIT Press, Cambridge, MA, 1990.

[K-74] A. V. KARZANOV, Determining the maximal flow in a network by the method of preflows, *Soviet Math. Dokl.* **15** (1974), 434–437.

[LF-80] R. E. LADNER AND M. J. FISCHER, Parallel prefix computation, *J. Assoc. Comput. Mach.* **27** (1980), 831–838.

[MPM-78] V. M. MALHOTRA, M. PRAMODH KUMAR, AND S. N. MAHESHWARI, An $O(|V|^3)$ algorithm for finding maximum flows in networks, *Inform. Process. Lett.* **7** (1978), 277–278.

[M-89] C. MARTEL, Review of four papers on network flow algorithms, *Comput. Rev.* **30**, No. 9 (1989), 488–490; reprint in *ACM-SIGACT News* **20**, No. 4 (1989), 26–29.

[PVW-83] W. PAUL, U. VISHKIN, AND H. WAGENER, Parallel computation on 2-3 trees, *RAIRO Theoret. Inform.* **17**, No. 4 (1983), 397–404; Parallel dictionaries on 2-3 trees *in* "Proceedings, 10th ICALP, Lecture Notes in Computer Science," Vol. 154, pp. 597–609, Springer-Verlag, New York/Berlin, 1983.

[O-88] J. B. ORLIN, A faster strongly polynomial minimum cost flow algorithm, *in* "Proceedings 20th ACM Symp. on Theory of Computing, 1988," pp. 377–387; Work Paper No. 3060-89-MS, Sloan School of Management, MIT, 1989.

[S-78] Y. SHILOACH, "An $O(nI \log^2 n)$ Max-Flow Algorithm," STAN-CS-78-702, Computer Science Department, Stanford University, 1978.

[SV-82] Y. SHILOACH AND U. VISHKIN, An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms* **3** (1982), 128–146.

[ST-83] D. D. SLEATOR AND R. E. TARJAN, A data structure for dynamic trees, *J. Comput. System Sci.* **26** (1983), 362–391.

[T-84] R. E. TARJAN, A simple version of Karzanov's blocking flow algorithm, *Oper. Res. Lett.* **2** (1984), 265–268.

[T-89] R. E. TARJAN, public presentation, Johns Hopkins University, November 1989.

[V-81] U. VISHKIN, "Synchronized Parallel Computation," D.Sc. thesis, Computer Science Department, Technion, Haifa, Israel, 1981. [Hebrew]