A PARALLEL ALGORITHM FOR FINDING A BLOCKING FLOW IN AN ACYCLIC NETWORK

Andrew V. Goldberg
Robert E. Tarjan

CS-TR-186-88

October 1988

A Parallel Algorithm for Finding a Blocking Flow in an Acyclic Network

*Andrew V. Goldberg**
Department of Computer Science
Stanford University
Stanford, CA 94305

Robert E. Tarjan†
Department of Computer Science
Princeton University
Princeton, NJ 08544
and
AT&T Bell Laboratories
Murray Hill, NJ 07974

October 1988

*Research partially supported by NSF Presidential Young Investigator Grant CCR-8858097, IBM Faculty Development Award, and ONR Contract N00014–88–K–0166. Part of this work was done while the author was at AT&T Bell Laboratories, Murray Hill, NJ 07974.

†Research partially supported by the National Science Foundation, Grant No. DCR-8605961, and the Office of Naval Research, Contract No. N00014-87-K-0467.

Abstract

We propose a simple parallel algorithm for finding a blocking flow in an acyclic network. On an n-vertex, m-arc network, our algorithm runs in $O(n \log n)$ time and $O(nm)$ space using an m-processor EREW PRAM. A consequence of our algorithm is an $O(n^2(\log n)\log(nC))$-time, $O(nm)$-space, m-processor algorithm for the minimum-cost circulation problem, on a network with integer arc capacities of magnitude at most C.

1 Terminology

In this paper we use the following definitions. Let $G = (V, E)$ be an acyclic directed graph with vertex set V of size n and arc set E of size m. For ease in stating time bounds, we assume that $m \geq n - 1$. Define $E^{-1} = \{(w, v) | (v, w) \in E\}$ and $E^+ = E \bigcup E^{-1}$. For any vertex w we denote by $E(w)$ the set of vertices *adjacent out from* w, $E(w) = \{x | (w, x) \in E\}$, and by $E^{-1}(w)$ the set of vertices *adjacent into* w, $E^{-1}(w) = \{v | (v, w) \in E\}$. Graph G is *layered* if each vertex v can be assigned an integer *layer* $L(v)$ such that $L(w) = L(v) + 1$ for every arc (v, w).

Graph G is a *network* if it has two distinguished vertices, a *source* s and a *sink* t, and a nonnegative real-valued capacity $u(v, w)$ on every arc (v, w). A *preflow* on a network is a nonnegative real-valued function f on the arcs such that $f(v, w) \leq u(v, w)$ for every arc (v, w) and $\sum_{v \in E^{-1}(w)} f(v, w) \geq \sum_{x \in E(w)} f(w, x)$ for every vertex $w \neq s$. The quantity $e(w) = \sum_{v \in E^{-1}(w)} f(v, w) - \sum_{x \in E(w)} f(w, x)$ is called the *excess* at vertex w. A preflow f is a *flow* if $e(w) = 0$ for every vertex $w \notin \{s, t\}$.

The *residual capacity* of an arc (v, w) with respect to a preflow f is $u_f(v, w) = u(v, w) - f(v, w)$. Arc (v, w) is *saturated* if $u_f(v, w) = 0$ and *residual* if $u_f(v, w) > 0$. A preflow is *blocking* if every path in G from s to t contains at least one saturated arc, *i.e.*, there is no path of residual arcs from s to t.

Our model of parallel computation is the exclusive-read, exclusive-write parallel random-access machine (EREW PRAM) [7]. We shall also briefly consider distributed computation models [10].

2 Perspective

The problem of finding a blocking flow in an acyclic network arises as a subproblem in computing maximum flows and in computing minimum-cost circulations. Specifically, Dinic [5] showed that the maximum flow problem can be solved by solving a sequence of $O(n)$ blocking flow problems on layered networks. We [12, 13, 14] have shown that the minimum-cost circulation problem can be solved by solving a sequence of $O(n \log(nC))$ blocking flow problems on acyclic but not necessarily layered networks. In the latter bound, C is the maximum absolute value of an arc cost; all arc costs are assumed to be integers.

Motivated by Dinic's discovery, several researchers have developed algorithms for finding a blocking flow in a layered network [2, 5, 8, 9, 16, 18, 21, 22, 23, 24]. Many of these algorithms, e.g. [5, 9, 16, 18, 22, 23, 24], work with the same asymptotic efficiency on arbitrary acyclic networks as on layered networks.

The asymptotically fastest known sequential algorithm is described in [14]; it runs in $O(m \log(n^2/m))$ time and $O(m)$ space, for an arbitrary acyclic network.

Of the cited algorithms, only one is a parallel algorithm, that of Shiloach and Vishkin [21], which runs in $O(n \log n)$ time and $O(n^2)$ space (U. Vishkin, private communication, 1986) using n processors. The Shiloach-Vishkin algorithm is stated for layered networks. Although we previously claimed that their method extends to arbitrary acyclic networks without loss of asymptotic efficiency[12, 13], this does not seem to be true; their running time analysis breaks down in the general case. Thus their algorithm cannot be used as an efficient subroutine in solving minimum-cost circulation problems.

Our goal in this paper is to devise a fast parallel blocking flow algorithm for arbitrary acyclic networks. In the next section we describe a method based on the concept of (flow) atoms; we call this method the *atomic* method. In Section 4 we give a parallel implementation of the atomic method. This implementation runs in $O(n \log n)$ time and $O(nm)$ space using m processors. As a corollary, we obtain an $O(n^2(\log n)\log(nC))$-time, $O(nm)$-space, m-processor parallel algorithm for the minimum-cost circulation problem. (See [12, 13, 14].)

3 The Atomic Method

In this section we describe a method for finding blocking flows in acyclic networks that is based on the concept of *atoms* (defined below). Atoms have been used previously in the analysis of maximum flow algorithms by Goldberg [11] and Cheriyan and Maheshwari [1].

Our general method is the same as that used by Karzarov [16] and later by others, e.g. [2, 8, 14, 21, 24]. The algorithm begins with a blocking preflow and moves flow excess through the network while maintaining a blocking preflow, until eventually this flow movement produces a blocking flow. The algorithm maintains a partition of the vertices into two states: *blocked* and *unblocked*. We call an arc (v, w) *admissible* if it is residual and w is unblocked. The algorithm blocks a vertex v when it discovers that none of the arcs leaving v is admissible; once v is blocked, every path from v to t contains a saturated arc. Excess on blocked vertices is returned from whence it came, by decreasing the flow on appropriate incoming arcs.

To keep track of the detailed flow movements, the algorithm maintains a partition of the flow excess into *atoms*. Consider a time during an execution of the algorithm. An atom is a maximal quantity of excess that has moved in exactly the same way so far. An atom a at a vertex v consists of an amount of excess denoted by *size*(a); the vertex v is denoted by *position*(a). An atom located at a vertex other than s or t is called *active*.

Associated with an atom a at a vertex v is a path of arcs in E^+ from s to v that the atom followed in arriving at v. This path is denoted by *trace*(a). Also associated with a is a simple path from s to v, denoted by *path*(a), of arcs in E through which the atom moved forward but not backward in the course of reaching v from s. The relationship between *trace*(a) and *path*(a) is that *path*(a) contains each arc (v, w) such that (v, w) but not (w, v) is on *trace*(a). The intuition behind the algorithm is that

```
procedure Process-Atom(a).
begin
    w ← position(a);
    if w is unblocked then
        if ∃ (w, x) : u_f(w, x) > 0 and x is unblocked then begin
            if size(a) > u_f(w, x) then begin
                [split a]
                create a new atom a';
                path(a) ← path(a');
                size(a') ← size(a) − u_f(w, x);
                size(a) ← u_f(w, x);
            end;
            position(a) ← x;
            append (w, x) to path(a);
            u_f(w, x) ← u_f(w, x) − size(a);
        end
        else mark w as blocked;
    if w is blocked then begin
        (v, w) ← last arc on path(a);
        position(a) ← v;
        delete (v, w) from path(a);
        move a to v;
        update path(a);
        u_f(v, w) ← u_f(v, w) + size(a);
    end;
end.
```

Figure 1: The Process-Atom procedure. Note that the flow is maintained implicitly as a difference between u and u_f.

each atom does a depth-first search from s in an attempt to reach t. The graph being searched changes dynamically as arcs become saturated and vertices become blocked.

During initialization, the algorithm saturates every arc (s, v) leaving the source, creating at each neighbor v of s an atom of size $u(s, v)$ and trace (s, v). At each iteration, the algorithm selects an active atom a and processes it as described in Figure 1. Let $w = position(a)$. If w is not blocked, the algorithm tries to move a forward along an arc with positive residual capacity. If no such arc exists, w becomes blocked. If there is such an arc, the algorithm picks one, say (w, x). If $size(a) > u_f(w, x)$, atom a is split into two parts. One part, of size equal to $size(a) - u_f(w, x)$, gets a new name a'. The other part, of size equal to $u_f(w, x)$, retains the name a. Atom a' remains at vertex w to be processed later; atom a moves to vertex x. Finally, if atom a has not moved (*i.e.*, vertex w is blocked), atom a is returned to the vertex, say v, from which it first reached w.

Note that an atom can move in two ways: forward from w to x or backward from w to v. In the former case, w is unblocked and $f(w, x)$ increases. In the latter case, w is blocked and $f(v, w)$ decreases. An atom can move backward from w to v only if at a previous time it moved forward from v to w. Thus the flow through an arc never becomes negative. During the course of the algorithm, for any arc

(w, x), the flow on (w, x) first increases, until x becomes blocked, after which the flow decreases.

Note that we have not specified the way in which we select an atom to be processed next. In the parallel implementation of the algorithm, all active atoms, and atoms arising from them by iterated splitting, are processed concurrently. In the sequential implementation, any constant-time selection rule leads to an $O(nm)$ time bound. For example, one can maintain the set of active atoms as a queue or a stack. Alternatively, at each vertex one can maintain a list of the atoms located at the vertex, and keep a queue or a stack of vertices with nonempty lists of atoms.

We begin our analysis of the algorithm by bounding the number of atoms.

Lemma 3.1 The total number of atoms created during an execution of the atomic algorithm is at most m.

Proof: We claim that each increase in the number of atoms corresponds to an arc saturation. Atoms created during initialization are charged to the saturation of the corresponding arcs. An atom created by splitting in procedure *Process-Atom* is charged to the saturation of the arc (w, x) in the same execution of the procedure. Thus the claim is true. Since each arc becomes saturated only once, the lemma is true. ∎

The next lemma gives the key property of the algorithm. Intuitively, the lemma holds because the trace of an atom is a partial traversal of a tree rooted at s.

Lemma 3.2 Consider an atom a at some time during execution of the algorithm. Then the length of the trace of a is at most $2n - 3$.

Proof: An atom a only moves backward from a vertex $w \notin \{s, t\}$ once w is blocked. Just after a moves backward from w, w is not on $path(a)$, and a never visits w again. It follows that, for each vertex $w \neq t$, $E \bigcap trace(a)$ contains at most one arc of the form (v, w); and, for each vertex $w \notin \{s, t\}$, $E^{-1} \bigcap trace(a)$ contains at most one arc of the form (w, v). This gives a bound of $2n - 3$ on the length of $trace(a)$. ∎

We define *phases* of the algorithm as follows. Initialization is phase 1. Phase i for $i > 1$ begins at the end of phase $i - 1$ and ends as soon as every atom that existed at the end of the phase $i - 1$, and every atom created by splitting since the end of phase $i - 1$, has moved at least one step. Since every atom moves (either forward of backward) at least once during each phase, we have the following corollary, which is crucial for the analysis of parallel versions of the atomic method.

Corollary 3.3 The number of phases during an execution of the algorithm is at most $2n - 3$.

To obtain an efficient implementation of the algorithm, we maintain the path of each atom as a stack of arcs.[1]. When an atom moves forward along an arc, the arc is pushed on top of the stack. To move an atom backward, we move it to the tail vertex of the top-of-stack arc and pop the stack.

Using stacks allows the algorithm to move atoms forward and backward in constant time. Splitting an atom, however, requires copying a stack. For ordinary stacks, this requires linear time. A very simple

[1]If the network has no multiple arcs, it is sufficient to maintain stacks of vertices on the paths.

implementation of *persistent stacks* [19] (see also [6]) allows the copy operation, as well as the push and pop operations, to be done in constant time. In combination with Lemmas 3.1 and 3.2, this fact gives the following result.

Theorem 3.4 The atomic algorithm, implemented using persistent stacks, runs in $O(nm)$ time.

4 A Parallel Implementation

In this section we describe a parallel implementation of the atomic method. The parallel implementation works in pulses; at each pulse, every atom, including those arising by splitting, moves either forward or backward or both. Thus each pulse completes at least one phase, where a phase is as defined in Section 3.

The parallel implementation consists of the following four steps:

Step 1 (initialize). For each arc (s,v), set $f(s,v) = u(s,v)$. Create an atom at v of size $f(s,v)$ and having stack containing only (s,v). For each arc (v,w) with $v \neq s$, set $f(v,w) = 0$. Block vertex s and unblock all other vertices.

Step 2 (push flow forward). For each unblocked vertex $w \notin \{s,t\}$, in parallel, do the following.

Arbitrarily order the atoms at w, say $a_1, a_2, \ldots, a_k$, and the admissible arcs (w,x), say (w,x_1), $(w,x_2), \ldots$, (x,w_l). For $1 \leq j \leq k$, compute a *cumulative size* $S(j) = \sum_{i=1}^{j} size(a_i)$. For each $1 \leq j \leq l$, compute a *cumulative residual capacity* $R(j) = \sum_{i=1}^{j} u_f(w,x_i)$. Assign the atoms a_i to the admissible arcs (w,x_j) as follows:

1. If $S(i) - size(a_i) \geq R(j) - u_f(w,x_j)$ and $S(i) \leq R(j)$, assign all of atom a_i to (w,x_j).
2. If $S(i) - size(a_i) \geq R(j) - u_f(w,x_j)$ and $S(i) > R(j)$, assign an amount $R(j) - S(i) + size(a_i)$ of atom a_i to (w,x_j).
3. If $S(i) - \text{size}(a_i) < R(j) - u_f(w,x_j)$ and $S(i) > R(j)$, assign an amount $u_f(w,x_j)$ of atom a_i to (w,x_j).
4. If $S(i) - size(a_i) < R(j) - u_f(w,x_j)$ and $S(i) \leq R(j)$, assign an amount $S(i) - R(j) + u_f(w,x_j)$ of atom a_i to (w,x_j).

(This assignment associates with each admissible arc a total amount of excess less than or equal to its residual capacity. At most one such arc receives an amount that is positive but less than its residual capacity. The total amount assigned to admissible arcs equals the minimum of the excess at w and the sum of the residual capacities of the admissible arcs (w,x_j).)

Split any atom assigned to more than one arc into two or more atoms, one per assigned arc, each of size equal to the amount of the original atom assigned to the arc. Each of the new atoms inherits the assignment of the corresponding amount of the old atom, as well as the a copy of the stack of the old atom.

For each admissible arc (w,x_j), increase $f(w,x_j)$ by the sum of the sizes of the atoms assigned to (w,x_j), and move each such atom to x_j, pushing (w,x_j) to its stack. If all arcs (w,x_j) are now saturated, mark w to be blocked. (Do not block w yet.)

Step 3 (block vertices). Block every vertex marked to be blocked in Step 2.

Step 4 (return flow). For each blocked vertex $w \notin \{s, t\}$, in parallel, do the following:

For each atom a at w, let (v_a, w) be the top arc on $stack(a)$. Pop $stack(a)$. Decrease $f(v_a, w)$ by $size(a)$ and move a to v.

Step 5 (loop). If every atom is at s or t, stop. Otherwise, go to Step 2.

By using standard techniques of parallel computation [15], including fast sorting [3], parallel prefix computations [17], and computations based on complete binary trees [25], one can implement each step of the algorithm to run in $O(\log n)$ time on an m-processor PRAM. (Lemma 3.1 implies that only m processors are necessary.) The details are routine and we omit them; the reader can refer to [12, 21] for more details on how to use these techniques to implement flow algorithms.

The running time of the entire algorithm is then $O(n \log n)$ by Corollary 3.3. The space required is dominated by the space for the paths of atoms, which is $O(nm)$.

5 Distributed Implementation

The atomic method has a natural implementation in a distributed model of computation, due to the robustness of the order in which active atoms are processed. By Lemma 3.2, a straightforward implementation of the atomic algorithm on either a synchronous or an asynchronous distributed model of computation [10] works in $O(n)$ message-passing rounds using $O(nm)$ messages. We can thread the persistent stacks representing the paths through the vertices to obtain an $O(m)$ space bound per vertex.

Recall that we would like to use the blocking flow algorithm as a subroutine in our minimum-cost circulation method [12, 13, 14]. In order to do this, we need to add termination detection to our distributed algorithm (so that the processors know when to start the next stage of the minimum-cost circulation algorithm). The termination detection can be obtained without increasing the asymptotic time bounds by using the technique of Dijkstra and Scholten [4] for detecting termination of diffusing computations (a simpler termination detection technique specific to minimum-cost circulation algorithms is discussed in [12]). The Dijkstra-Scholten technique works for algorithms with a single initiator. This is not a problem for the blocking flow algorithm described in this paper, since the algorithm is initiated by the source processor. The version of the problem that comes up in the execution of the minimum-cost circulation algorithm, however, has several capacitated sources instead of a single uncapacitated source. Therefore, we need to construct a spanning tree in the network and select a leader before running the minimum-cost circulation algorithm. Even in the asynchronous model, this preprocessing can be done in $O(n \log n)$ time, which is dominated by the $O(n^2 \log(nC))$ running time of the minimum-cost circulation algorithm.

Note that the above bounds for distributed computation are not very good from the theoretical viewpoint. We do just as well by sending all the information about the network to a single vertex and letting it do all the computation. In practice, however, our distributed algorithm should be more

efficient than such a centralized computation.

6 Concluding Remarks

In conclusion, we would like to discuss some open questions related to the problems studied in this paper.

The parallel complexity of the blocking flow problem (in layered, acyclic, and general networks) is wide open. This problem is not known to be in NC; nor is it known to be P-complete. Resolving either of these questions seems to be hard. A possibly simpler question is whether an $O(n^\epsilon)$-time blocking flow algorithm for $0 < \epsilon < 1$ exists.

Orlin's minimum-cost circulation algorithm [20], implemented using the best parallel shortest path algorithm currently known, solves the minimum-cost circulation problem in $O(m \log^3 n)$ time using $n^3/\log n$ processors. Although for most possible values of n, m, and C, this time bound is better then the time bound achieved by our minimum-cost circulation algorithm discussed in Section 2, our algorithm is more practical since it uses only m processors.

There are some obvious inefficiencies in our algorithm. Though the running time is faster than that of our sequential algorithm [14] by a factor of $m \log(n^2/m)/(n \log n)$, the total work done by the algorithm (the product of the running time and the number of processors) is $O(n^2 m \log n)$, a factor of $n \log n / \log(n^2/m)$ worse than that of our sequential algorithm. The sequential algorithm uses much more complicated data structures, however. If only simple data structures are used, the running time bound of our sequential algorithm increases by a factor of $m/(n \log(n^2/m))$. Even then, the total work done by the parallel algorithm is greater by a factor of $(m \log n)/n$. The atomic method can be improved by combining atoms that are at the same vertex at the same time and moving them forward together, thereby reducing the number of forward flow pushes. Also, if some of the excess at a vertex v is to be returned from v, it does not matter which part of the excess is selected for returning, since there is only one kind of commodity involved. It is easy to design an improved algorithm based on these ideas, but we have been unable to obtain any improvement in our asymptotic resource bounds by doing so. The Shiloach-Vishkin result [21] suggests the possible existence of a blocking flow algorithm for acyclic networks running in $O(n \log n)$ time and $O(n^2)$ space using n processors. Finding such an algorithm, or disproving its existence, is a challenging open problem.

References

[1] J. Cheriyan and S.N. Maheshwari. Analysis of a preflow push algorithm for maximum netwrok flow. Unpublished manuscript, Department of Computer Science and Engineering, Indian Institute of Technology, New Deli, India, 1987.

[2] R. V. Cherkasky. Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations. *Mathematical Methods of Solution of Economical Problems*, 7:112–125, 1977. (In Russian).

[3] R. Cole. Parallel merge sort. In *Proc. 27th IEEE Annual Symposium on Foundations of Computer Science*, pages 511–516, 1986.

[4] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Inform. Proc. Letters*, 11:1–4, 1980.

[5] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. J. Comput. Sys. Sci., to appear.

[7] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[8] Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Informatica*, 14:221–242, 1980.

[9] Z. Galil and A. Naamad. An $O(EV\log^2 V)$ algorithm for the maximal flow problem. *J. Comput. System Sci.*, 21:203–217, 1980.

[10] R. G. Gallager, P. A. Humblet, and P. M Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, 1983.

[11] A. V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.

[12] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., Cambridge, MA, 1987).

[13] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 7–18, 1987.

[14] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. Math. of Oper. Res., to appear.

[15] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared memory machines. Technical Report UCB/CSD 88/408, Computer Science Division (EECS), University of California, Berkeley, CA, 1988.

[16] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dok.*, 15:434–437, 1974.

[17] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. Assoc. Comp. Mach.*, 27:831–838, 1980.

[18] V. M. Malhotra, M. Pramodh Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inform. Process. Lett.*, 7:277–278, 1978.

[19] E. W. Myers. An applicative random-access stack. *Inform. Proc. Letters*, 17:241–248, 1983.

[20] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 377–387, 1988.

[21] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[22] D. D. Sleator. An $O(nm \log n)$ algorithm for maximum network flow. Technical Report STAN-CS-80-831, Computer Science Department, Stanford University, Stanford, CA, 1980.

[23] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[24] R. E. Tarjan. A simple version of Karzanov's blocking flow algorithm. *Operations Research Letters*, 2:265–268, 1984.

[25] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14:862–874, 1985.