Information Processing Letters 31 (1989) 157-161
North-Holland 8 May 1989

THE PARALLEL COMPLEXITY OF FINDING A BLOCKING FLOW IN A 3-LAYER NETWORK

J. CHERIYAN

Computer Science Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Colaba, Bombay-5, India

S.N. MAHESHWARI

Department of Computer Science & Engineering, Indian Institute of Technology, Hauz Khas, New Delhi-16, India

Communicated by David Gries
Received 23 June 1988
Revised 4 October 1988

Keywords: Blocking flow, lex-first blocking flow, log-space complete for P, max-flow problem, parallel algorithm, random NC

1. Introduction

We study the parallel complexity of the blocking flow problem. Let $G(V, E)$ be a network (i.e. a weighted digraph) with two distinguished vertices, a source s and a sink t, and a positive real-valued capacity $c(v \to w)$ on every edge $v \to w$. (For convenience define $c(v \to w) = 0$ if $v \to w$ is not an edge.) A flow on G is a real-valued function f on vertex pairs with the following two properties:

(1) *Capacity constraint*: $f(v \to w) \leqslant c(v \to w)$ for all vertex pairs v, w. If $v \to w$ is an edge such that $f(v \to w) = c(v \to w)$, we say the flow *saturates* $v \to w$.

(2) *Flow conservation*: $\Sigma_u f(u \to v) = \Sigma_w f(v \to w)$ for every vertex v other than s and t.

The value $|f|$ of a flow is the net flow into the sink, $\Sigma_v f(v \to t) - \Sigma_w f(t \to w)$. A flow f is a *blocking flow* if every path from source s to sink t contains a saturated edge (such a flow is called a maximal flow in [4]). The value of a blocking flow cannot be increased by pushing additional flow along any path in G, although it may be possible to increase the flow value by rerouting, i.e. decreasing the flow on some edges and increasing it on others. The *blocking flow problem* is to find a blocking flow. The *max-flow problem* is to find a flow of maximum value. A max-flow can be found by iterating any blocking flow algorithm at most $|V|$ times (see [6,4] for further details).

Shiloach and Vishkin [5] gave an $O(|V| \times (\log|V|)^2)$ time parallel algorithm for finding a blocking flow in an acyclic network, thereby giving an $O(|V|^2(\log|V|)^2)$ algorithm for max-flow. Subsequently, Goldschlager, Shaw and Staples [3] showed that the max-flow problem is log-space complete for P, so it is unlikely to be in NC. However, the parallel complexity of the blocking flow problem remains an interesting open question. We resolve this for a restricted class of acyclic networks called 3-layer networks (described below).

The lex-first blocking flow in an acyclic network is defined as follows. A linear ordering is imposed on the vertices of the network, and the adjacency list of each vertex is sorted according to this order. The *lex-first blocking flow* is the flow that results when the sequential dfs blocking flow algorithm (i.e. Dinic's algorithm [2,6]) is run on the acyclic network with adjacency lists ordered as above.

The first result here shows that the problem of finding the lex-first blocking flow is log-space complete for P. This indicates why it is difficult to improve the parallel blocking flow algorithm in [5], and it also suggests that if the time bound is to be improved, then a new approach should be used.

We then consider a restricted class of acyclic networks. A *layered network* is an acyclic network in which all source-to-sink paths have the same length, and the *length* of a layered network is the length of a source-to-sink path. We refer to a layered network of length 3 as a 3-layer network. The problem of finding the lex-first blocking flow in a 3-layer network is also shown to be log-space complete for P. However, we develop a random NC algorithm for finding an arbitrary blocking flow in a 3-layer network. The novel feature of this algorithm is that it makes crucial use of the edge capacities. This algorithm uses $O(|V|^2)$ processors, $O((\log|V|)^3)$ time and $O(|V| \times (\log|V|)^2)$ random bits.

We use the CREW PRAM model, in which concurrent reads from the same memory location are allowed but concurrent writes to the same memory location are disallowed.

We have omitted some proofs; the omitted details are given in [1].

2. Lexicographically first blocking flow is log-space complete for P

Our approach for showing that the lex-first blocking flow problem is log-space complete for P is based on the construction used in [3] to show that the max-flow problem is log-space complete for P. This construction reduces a problem known to be log-space complete for P, namely monotone circuit value problem with fan-out 2 (abbreviated MCV2), to the max-flow problem.

Given a monotone circuit C with fan-out 2, Goldschlager et al. [3] construct a flow network N_0. Then they define a *simulating flow pattern* f_s, which they show to be a max-flow in N_0; and they show that $|f_s|$ is odd iff the circuit C computes true. We shall say that a flow f satisfies the *simulating flow conditions* if for every edge $v_i \to t$ entering t, the flow $f(v_i \to t)$ equals the flow $f_s(v_i \to t)$ given by the simulating flow pattern. Clearly, if f satisfies the simulating flow conditions, then $|f| = |f_s|$ = value of any max-flow in N_0.

2.1. Theorem. *The problem of finding the value (modulo 2) of the lex-first blocking flow in f in a given network N is log-space complete for* P.

Proof. We shall reduce MCV2 to lex-first blocking flow. Given a monotone circuit C that is an instance of MCV2, we construct a network N that is the same as the network N_0 constructed in [3] except that all edges entering source s are removed. Notice that N is acyclic, and that the value of a max-flow in N equals the value of a max-flow in N_0. The lexical ordering of the vertices of N entails that s and t come first, followed by the remaining vertices in the order $v_n, \ldots, v_1, v_0$.

Claim. *In network N, which corresponds to circuit C, the lex-first blocking flow f satisfies the simulating flow conditions.*

It follows from this claim that $|f|$ equals the value of a max-flow in N_0. Thus, the theorem follows from this claim because the value of the lex-first blocking flow in network N is odd iff the output gate of the circuit C computes true.

The claim is proved by induction on the number of gates in the circuit C. □

The following lemma shows that the value of a max-flow in any (possibly cyclic) network can be found by computing the value of a max-flow in a suitably constructed 3-layer network. This construction was originally used by Wagner (see [4, p. 149]) to show that the minimum-cost flow problem is reducible to the Hitchcock transportation problem.

2.2. Lemma. *Given a network N with edge capacities $c(v \to w)$ in which the value of a max-flow is* maxval(N), *a 3-layer network N′ can be constructed such that the value of a max-flow in N′ equals the sum of* maxval(N) *plus the sum of all the edge capacities of N, i.e.*

$$\text{maxval}(N') = \text{maxval}(N) + \sum_{v \to w} c(v \to w).$$

This construction can be done using logarithmic work space.

2.3. Theorem. *The problem of finding the value (modulo 2) of the lex-first blocking flow f in a 3-layer network N' is log-space complete for* P.

3. A random NC algorithm for finding a blocking flow in a 3-layer network

Although the problem of finding the lex-first blocking flow in a 3-layer network $G(V, E)$ is log-space complete for P and hence unlikely to be in NC, we show that there is a random NC algorithm for finding an arbitrary blocking flow in a 3-layer network.

In the 3-layer network $G(V, E)$ let $U=\{u_1, u_2, \ldots, u_m\}$ be the set of neighbours of source s and let $Y=\{v_1, v_2, \ldots, v_n\}$ be the set of neighbours of sink t. Notice that the vertices in $U \cup Y$ induce the bipartite subgraph of G. First we consider the restricted class of simple 3-layer networks. In a *simple 3-layer network*, each edge $u_i \to v_j$ in the bipartite graph has infinite capacity and the degree $d(u_i)$ of each vertex u_i lies between 2^k and $2^{k+1}-1$ for some integer k.

Let the capacity of edge $s \to u_i$ be denoted by $c(u_i)$ and the capacity of edge $v_j \to t$ be denoted by $c(v_j)$. An edge $u_i \to v_j$ is said to be *blocked* if the total incoming flow to vertex v_j is $\geqslant c(v_j)$.

In order to obtain a poly-logarithmic bound on the time taken by the algorithm, we must eliminate a constant fraction of the edges in $G(V, E)$ during one iteration. An obvious approach that does not work is as follows. Suppose each vertex u_i pushes $c(u_i)/d(u_i)$ units of flow along each edge emanating from it. Consider an edge $u_i \to v_j$; either this edge remains unblocked or it becomes blocked. In the latter case, if the total incoming flow at v_j is more than $c(v_j)$, then some flow may have to be *reflected* back to u_i along edge $u_i \to v_j$. This reflected flow causes the essential difficulty in obtaining a parallel blocking flow algorithm for a simple 3-layer network.

Let us say that a vertex v_j belonging to Y is *small* if

$$c(v_j) \leqslant \sum_{u_i \to v_j} (c(u_i)/d(u_i));$$

v_j is *big* if

$$c(v_j) \geqslant \sum_{u_i \to v_j} 2(c(u_i)/d(u_i));$$

and v_j is *medium* otherwise.

Consider the approach of pushing $c(u_i)/d(u_i)$ units of flow along each edge emanating from u_i for each vertex u_i. Suppose a particular vertex u_i has a constant fraction (say $\frac{1}{8}$) of small neighbours. Then this approach would still succeed for u_i because enough edges would be blocked and hence these edges need not be considered any further. Also, if u_i has at least as many big neighbours as it has small neighbours, the approach would succeed for u_i because any reflected flow can be apportioned equally among the big neighbours in the second step of the algorithm. But when neither of the above two situations holds, we have to resort to randomization in order to apportion flow properly among the neighbours of u_i.

The medium vertices v_j are randomly partitioned into two classes (black and white) of roughly equal size by each medium vertex tossing an unbiased coin. Consequently, if a vertex u_i has many medium neighbours, then with high probability it has at least $\frac{1}{4}d(u_i)$ black medium neighbours and at least $\frac{1}{4}d(u_i)$ white medium neighbours.

Consider a vertex u_i that satisfies the last condition. u_i pushes $2c(u_i)/d(u_i)$ units of flow to a chosen set of $\frac{1}{4}d(u_i)$ white medium neighbours; it pushes zero flow to a chosen set of $\frac{1}{4}d(u_i)$ black medium neighbours and it pushes $c(u_i)/d(u_i)$ units of flow to its remaining neighbours. If this apportionment of flow blocks a constant fraction, say $\frac{1}{8}$, of the edges emanating from u_i, then we have succeeded. Otherwise, the total amount of flow reflected back to u_i is at most $(2c(u_i)/d(u_i))$ (number of blocked edges emanating from u_i) $\leqslant \frac{1}{4}c(u_i)$, because the amount of reflected flow along an edge $u_i \to v_j$ is $\leqslant 2c(u_i)/d(u_i)$. All the reflected flow at u_i is apportioned equally among the chosen set of black medium neighbours and it is easily seen that none of the black medium neighbours gets blocked. This follows because a black medium vertex, say v_j, receives at most

$$\sum_{u_i \to v_j} (c(u_i)/d(u_i)) < c(v_j)$$

units of incoming flow during one iteration. Thus we are done with vertex u_i since all the incoming flow $c(u_i)$ has been pushed out. The detailed procedure for one iteration of the algorithm is given in Appendix A.

The correctness, and time and processor complexity of one iteration of the above algorithm is given by the following lemmas.

3.1. Lemma. *The above algorithm is correct, i.e. each iteration computes a flow f that satisfies the capacity constraints and the flow conservation conditions.*

3.2. Lemma. *Let vertex u_i have less than $\frac{1}{8}d(u_i)$ small neighbours and less than $\frac{1}{8}d(u_i)$ big neighbours. Let p be the probability that u_i has at least $\frac{1}{4}d(u_i)$ black medium neighbours and has at least $\frac{1}{4}d(u_i)$ white medium neighbours. Then p is at least $\frac{1}{2}$, provided that $d(u_i)$ is at least 64.*

Proof. Clearly, u_i has at least $\frac{3}{4}d(u_i)$ medium neighbours. Note that the colour of neighbour v_j is independent of the colour of neighbour v_q, $j \neq q$, so the probability that there are exactly k black neighbours in a set of N medium neighbours is given by the binomial distribution. The proof is completed by upper-bounding the tail of the binomial distribution. □

3.3. Lemma. *The expected number of edges eliminated by (one iteration of) the above algorithm is at least $|E|/32$, where the given 3-layer network has $|E|$ edges.*

Proof. Consider a vertex u_i. If u_i either has at least $\frac{1}{8}d(u_i)$ small neighbours or has more big neighbours than small neighbours, then at least $\frac{1}{8}d(u_i)$ edges are eliminated. Now suppose that neither of the above two situations holds, i.e. u_i has at least $\frac{3}{4}d(u_i)$ medium neighbours.

If u_i has at least $\frac{1}{4}d(u_i)$ black medium neighbours and at least $\frac{1}{4}d(u_i)$ white medium neighbours, then either all the reflected flow is pushed to the sink and hence all the edges emanating from u_i are eliminated or $\frac{1}{8}d(u_i)$ edges are eliminated.

The lemma now follows from the previous lemma and the assumption that for each u_i,

$$2^k \leqslant d(u_i) \leqslant 2^{k+1} - 1. \quad \square$$

3.4. Lemma. *The above algorithm uses $O(nm)$ processors and $O(\log nm)$ time and n random bits (for one iteration) on a CREW PRAM.*

Proof. Observe that each step uses at most nm processors and either $O(\log n)$ time or $O(\log m)$ time. □

Instead of developing the full blocking flow algorithm for simple 3-layer networks we shall go directly to general 3-layer networks where the capacity of an edge $u_i \to v_j$ in the bipartite graph may not be infinite. Recall that the algorithm in Appendix A pushes at most $2c(u_i)/d(u_i)$ units of flow along an edge $u_i \to v_j$. We first eliminate low capacity edges in the bipartite graph as follows. Suppose a vertex u_i has at most $\frac{1}{4}d(u_i)$ edges of capacity less than $4c(u_i)/d(u_i)$ emanating from it (the other case is easily handled). Then u_i eliminates all these edges by pushing flow equal to its capacity along each such edge. Possibly all the flow pushed out by u_i may be reflected back to it, but even then the algorithm in Appendix A will push at most

$$8c(u_i)/(3d(u_i)) < 4c(u_i)/d(u_i)$$

units of flow along an edge $u_i \to v_j$. (Note that during the elimination of low capacity edges, a vertex v_j is not deleted even if the incoming flow to it is $c(v_j)$ or more.) The remaining edges in the bipartite graph have sufficiently high capacity that we may consider them to have infinite capacity when we run the algorithm in Appendix A.

Initially, we consider vertices u_i having $d(u_i) \geqslant \frac{1}{2}n$, and repeatedly run the algorithm in Appendix A on the subnetwork induced by these vertices and the vertices in $\{s\} \cup Y \cup \{t\}$ until enough edges are eliminated so that each vertex u_i has $d(u_i) < \frac{1}{2}n$. Then we consider vertices u_i in the resulting network that have $d(u_i) \geqslant \frac{1}{4}n$, and so on. Eventually, all the edges in the bipartite graph are eliminated and we have found a blocking flow.

3.5. Theorem. *The above-mentioned algorithm finds a blocking flow in a given 3-layer network. It uses $O(nm)$ processors and $O((\log nm)^2)$ iterations, on a CREW PRAM, where each iteration takes $O(\log nm)$ time and n random bits.*

Appendix A

Algorithm Simple-3-layer-network-blocking-flow-iteration

Input: Simple 3-layer network.

Output: Flow that eliminates a constant fraction of the edges (on the average) in the given network.

```
(1) classify each vertex $v_j$ belonging to $Y$ as small, big or medium;
(2) for each medium vertex $v_j$ do
    generate a random unbiased bit and colour $v_j$ black if the bit is zero and white otherwise;
(3) for each vertex $u_i$ do
    if $u_i$ has at least $\frac{1}{4}d(u_i)$ black medium neighbours and at least $\frac{1}{4}d(u_i)$ white medium neighbours
    then begin
        choose a set $B(u_i)$ of $\frac{1}{4}d(u_i)$ black medium neighbours and a set $W(u_i)$ of $\frac{1}{4}d(u_i)$ white medium neighbours;
        push $2c(u_i)/d(u_i)$ units of flow to each neighbour in $W(u_i)$, zero flow to each neighbour in $B(u_i)$, and $c(u_i)/d(u_i)$ units of flow to each of the remaining neighbours
    end
    else begin
        $B(u_i) := \emptyset$; $W(u_j) := \emptyset$;
        push $c(u_i)/d(u_i)$ units of flow to each neighbour
    end;
(4) for each vertex $v_j$ do
    if the incoming flow $\leqslant c(v_j)$
    then push all the incoming flow to sink $t$
    else push $c(v_j)$ units of flow to $t$ and reflect back the remaining flow to the vertices $u_i$ that sent the flow;
    (* note that the amount of flow reflected back by $v_j$ along edge $u_i \to v_j$ lies between zero and the amount of flow pushed by $u_i$ along $u_i \to v_j$; thus, the amount of reflected flow along $u_i \to v_j$ is $\leqslant 2c(u_i)/d(u_i)$ *);
(5) for each vertex $u_i$ do
        let $r(u_i)$ be the total reflected flow at $u_i$;
        $B'(u_i) := B(u_i) \cup \{v_j \mid v_j$ is a big neighbour$\}$;
    if $(r(u_i)/|B'(u_i)|) \leqslant (c(u_i)/d(u_i))$
    then push $r(u_i)/|B'(u_i)|$ units of flow to each neighbour in $B'(u_i)$ and
        reset $r(u_i)$, i.e. $r(u_i) := 0$;
    replace $c(u_i)$ by $r(u_i)$, i.e. $c(u_i) := r(u_i)$, and update $d(u_i)$.
(6) for each vertex $v_j$ do update $c(v_j)$.
```

Remark. The actual implementation of the above steps on a CREW PRAM is straightforward and can be done using balanced binary tree computations.

References

[1] J. Cheriyan, Algorithmic studies in graph connectivity and in network flows, Ph.D. Thesis, Department of Computer Science and Engineering, IIT, New Delhi, 1988.

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

[3] L.M. Goldschlager, R.A. Shaw and J. Staples, The maximum flow problem is log-space complete for P, *Theoret. Comput. Sci.* **21** (1982) 105–111.

[4] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).

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

[6] R.E. Tarjan, *Data Structures and Network Algorithms* SIAM, Philadelphia, PA, 1983).