Experimental Evaluation of a Parallel Max-Flow Algorithm

Goranka Nogo and Robert Manger

Department of Mathematics, University of Zagreb, Bijenicka 30, 10000 Zagreb, Croatia

The maximum flow problem has been studied for over forty years. One of the methods for solving this problem is the generic push-relabel algorithm. In this paper we develop a parallel version of this sequential algorithm. Our assumed model of computation is a shared-memory multiprocessor. We describe a concrete implementation of the algorithm based on the *PVM* package, and present the obtained numerical results.

Keywords: network, maximum flow problem, parallel algorithm, *PVM*, experiment

1. Introduction

To introduce the maximum flow problem, we need the following definitions. A *network* is a directed graph $G = (V, E)$ with a non-negative *capacity function* $u : E \to \mathbf{R}$. We assume that G has no multiple arcs. A *flow network* is a network with two distinguished nodes, the *source* s and the *sink* t.

A *pseudoflow* is a function $f : E \to \mathbf{R}$ that satisfies the following constraints:

1. $f(v, w) \leq u(v, w)$, $\forall (v, w) \in E$ (capacity constraint),
2. $f(v, w) = -f(w, v)$, $\forall (v, w) \in E$ (flow antisymmetry constraint).

We define the *excess* function $e_f : V \to \mathbf{R}$ by $e_f(v) := \sum_{w \in V} f(w, v)$, the net flow into v. We will say that a node v has an *excess* if $e_f(v)$ is positive. For a node v, we define the *conservation constraint* by $e_f(v) = 0$.

Given a pseudoflow f, the *residual capacity* function $u_f : E \to \mathbf{R}$ is defined by $u_f(v, w) := u(v, w) - f(v, w)$. The *residual graph* with respect to a pseudoflow f is given by $G_f = (V, E_f)$, where $E_f := \{(v, w) \in E : u_f(v, w) \geq 0\}$.

A *preflow* is a pseudoflow f such that the excess function is non- negative for all nodes other than s and t. A *flow* f on G is a pseudoflow satisfying the conservation constraints for all nodes except s and t. The *value* $|f|$ of a flow f is the net flow into sink $e_f(t)$. A *maximum flow* is a flow of maximum value. The maximum flow problem is that of finding maximum flow in a given network.

As a measure of the network size, we use n to denote the number of nodes, and m to denote the number of arcs. The *network density* is the ratio $\frac{m}{n^2}$.

2. Generic Algorithm

In this section we describe the generic algorithm developed by Goldberg and Tarjan (Goldberg *et al.*, 1988). First, however, we need the following definitions, in addition to the definitions of the previous section. Consider a flow network (G, u, s, t). Define E' to be the set obtained by reversing the arcs on E. The arcs from E' perform the excess return to the source. For a given preflow f, a *distance labelling* is a function d from the nodes to the non-negative integers such that $d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for all residual arcs (v, w). We say that a node v is *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$. Note that a preflow f is a flow if and only if there are no active nodes. An arc (v, w) is *admissible* if $(v, w) \in E_f$ and $d(v) = d(w) + 1$. We assume

that the network capacities are integers. The algorithm begins with a preflow f that is equal to the arc capacity on each arc leaving the source, and zero on all arcs not incident to the source, and with some initial labelling d. The simplest choice is $d(s) = n$ and $d(v) = 0, \forall v \neq s$. The algorithm then repetitively performs, in any order, the *update operations*, *push* and *relabel*, described in Fig. 2. When there are no active nodes, the algorithm terminates. A summary of the algorithm appears in Fig. 1.

procedure *generic(V, E, u)*
(*initialisation*)
for $(v, w) \in E$ **do begin**
  $f(v, w) \leftarrow 0$;
  if $v = s$ **then** $f(v, w) \leftarrow u(v, w)$;
  if $w = s$ **then** $f(v, w) \leftarrow -u(w, v)$;
end;
for $w \in V$ **do begin**
  $e_f(w) \leftarrow \sum_{(v,w)\in E} f(v, w)$;
  if $w = s$ **then** $d(w) \leftarrow n$ **else** $d(w) \leftarrow 0$;
end;
(*main loop*)
while $\exists$ an active node **do**
  select an update operation and apply it;
return(f);
end.

Fig. 1. The generic maximum flow algorithm

procedure *push(v, w)*
if v is active and (v, w) admissible **then**
  send $\min\{e_f(v), u_f(v, w)\}$ units of flow from v to w;
end.

procedure *relabel(v)*
if $u_f(v, w) = 0, \forall (v, w) \in E$ **then**
  $d(v) \leftarrow \min_{(v,w)\in E'_f}\{d(w)\} + 1$;
else if $d(w) \geq d(v), \forall (v, w) \in E$ **then**
  $d(v) \leftarrow \min_{(v,w)\in E_f}\{d(w)\} + 1$;
end.

Fig. 2 The update operations

The following theorem gives a bound of the number of update operations.

Theorem 1. *(Korte et al., 1990) The generic algorithm terminates after $O(mn^2)$ update operations.*

Let us note that Theorem 1 holds for any choice of an active node and any choice of an admissible arc. An actual implementation should specify these details.

3. Parallel Version

We now describe a parallel version of the generic algorithm. Our assumed model of computation is a shared-memory multiprocessor where no two processors are allowed to write into the same location simultaneously (Chrislow, 1997). The first step toward an efficient parallel implementation is to find a way of choosing admissible arcs. We need some data structures to represent the network and the flow. To each node v, a list of the incident nodes:

$$list(v) := list\_of\_successors \cup list\_of\_predecessors,$$

is attached in fixed but arbitrary order.

The procedure *pass_list*() is applicable to an active node v. It processes the $list(v)$. We have two passes through the $list(v)$ starting from the first node w in the $list(v)$. If a pushing operation is applicable to the arc (v, w), the procedure *pass_list* pushes the excess of v through the arc (v, w). If not, w is replaced with the next node in the $list(v)$ or, if w is the last node, the procedure relabels v. After relabelling, starting again from the first node, the procedure pushes the excess at v. The procedure stops after the second pass, or when $e_f(v) = 0$.

Let l_j be a list consisting of all active nodes with labels j sorted by decreasing value of e_f. The set of active nodes is a list $\mathcal{L} := l_i \cup l_{i-1} \cup ... \cup l_0$ in fixed order (some lists may be empty). The procedure *remove_node*(v) removes the first node v from $\mathcal{L}$, and calls the procedure *pass_list*. Let p be the number of processors, $|\mathcal{L}|$ the length of $\mathcal{L}$, and $\{v_1, ..., v_k\}$ the first k elements (nodes) of $\mathcal{L}$, where $k = \min\{p, |\mathcal{L}|\}$. The algorithm begins just like the generic algorithm with a preflow f, and with some initial labelling d. The list $\mathcal{L}$ contains all nodes incident to the source. The parallel algorithm is described in Fig. 3.

procedure *parallel_max_flow_algorithm*(V, E, u)
(*initialisation*)
initialisation of f;
initialisation of e_f and d;
initialisation of $\mathcal{L}$;

```
(*main loop*)
while L ≠ ∅ do begin
    evaluation of k and {v_1, ..., v_k};
    (*main parallel loop*);
    for j ∈ {1, ..., k} do in parallel
        remove_node(v_j);
    parallel update of e_f and d;
    parallel update of L;
end;
return(f);
end.
```

Fig. 3. The parallel max-flow algorithm

Parallel update of e_f, d and $\mathcal{L}$ can be done in many different ways. One possibility is simply to distribute update operations among parallel processes and to rely on proper synchronisation (mutual exclusion) of any two operations that would try to access the same structure at the same time. An actual implementation of the algorithm should provide appropriate synchronisation mechanisms.

Results regarding the correctness and complexity of the considered parallel max-flow algorithm have been stated and proved in (Nogo, 1998). The following two theorems, which give bounds on time and space complexity, have been obtained.

Theorem 2. *Let $p = \lceil \frac{m}{n} \rceil$. Our parallel algorithm solves the max-flow problem in $O(n^3)$ time using $O(m)$ memory per processor.*

Theorem 3. *Suppose that $p \geq n$. Our parallel generic algorithm solves the max-flow problem in $O(n^2)$ time using $O(m)$ memory per processor.*

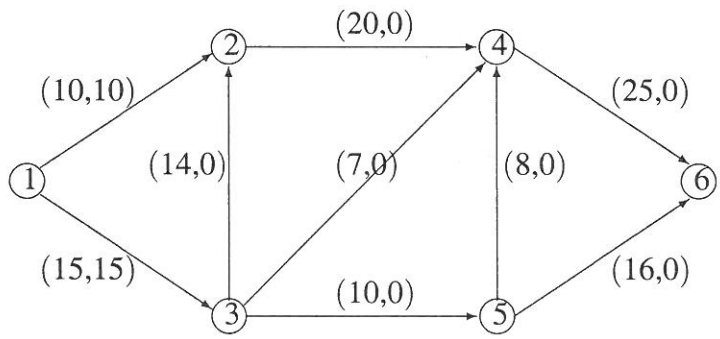
According to the theoretical estimates quoted above, our parallel algorithm uses less memory than the Shiloach-Vishkin and Goldberg algorithms respectively (Shiloah *et al.*, 1982; Goldberg, 1985). Also, our algorithm should run faster in all situations, except for high-density networks. For more details, see (Nogo, 1998).

4. An Example

To illustrate our parallel algorithm, let us consider the network in Fig. 4. Capacities $u(v, w)$ and initial pseudoflows $f(v, w)$ are given as arc labels. The initial values of the excess function e_f and of the labelling function d are specified in a table below the graph.

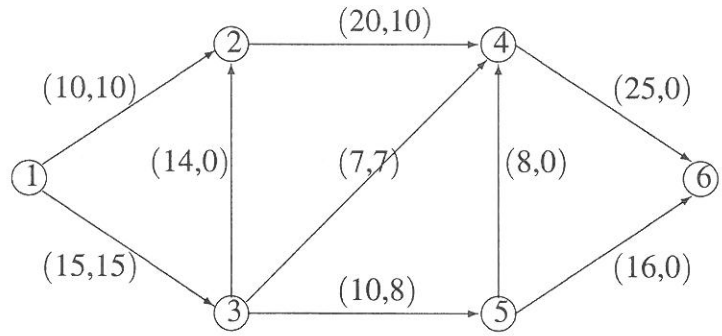
Suppose that our parallel machine has $p = 2$ processors. Assume also that in the first pass of the **while** loop the nodes $v_1 = 3$ and $v_2 = 2$ have been chosen as active nodes and processed in parallel. Then, after the first pass, the situation looks as shown in Fig. 5.

The next Fig. 6. describes the situation after the second pass of the **while** loop. Now the nodes $v_1 = 4$ and $v_2 = 5$ have been chosen and processed in parallel.



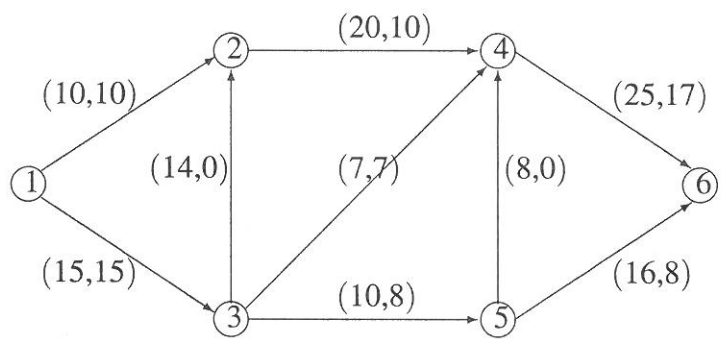
	1	2	3	4	5	6
e	–	10	15	0	0	0
d	6	0	0	0	0	0

Fig. 4. A sample network given as input to our algorithm



	1	2	3	4	5	6
e	–	0	0	17	8	0
d	6	1	1	0	0	0

Fig. 5. The sample network after the first pass of the algorithm



	1	2	3	4	5	6
e	–	0	0	0	0	25
d	6	1	1	1	1	0

Fig. 6. The sample network after the second pass of the algorithm

Since all nodes except $s = 1$ and $t = 6$ are now inactive, the algorithm terminates. The flow f given by Fig. 6. is a maximum flow.

5. *PVM* Implementation

The *PVM* software enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. In order to implement our parallel algorithm, we used the master-slave (or host-node) model of computation and data parallelism. According to this model a separate control program, termed the master, is responsible for process spawning, initialisation, collection and display of results. The slave programs receive initial data sets, do the actual computation, and send the resulting partial solution back to the master process. After sending the initial data, the master process simply waits for the results. When the results arrive, they are integrated into the solution arrays. Data parallelism means that all the slave processes are the same. The master-slave model described above involves no communication among the slaves. The problems like simultaneous writing into the same location, or simultaneous writing and reading from the same location, were solved using the local synchronisation method (Geist

et al., 1994). The master and slave programs were written in the *C* language. Test data were pseudo-random flow networks created in Mathematica (Wolfram, 1996). The network capacities were pseudo-random integers between 1 and 100. The network, the capacities and the flow were represented as one-dimensional arrays just like e_f, d and $\mathcal{L}$.

For experimental evaluation of our parallel algorithm, we used the following computers:

- Sun Ultra Enterprise 3000 server with two processors (master and first two slaves);
- HP 9000/E55 server (third slave);
- HP 9000/845SE server (fourth slave) and,
- HP Apollo 9000/712 workstation (fifth slave).

In addition to our parallel algorithm, we also implemented the original generic sequential algorithm. It was executed on the Sun Ultra Enterprise 3000 server.

6. Numerical Results

First we describe few concepts that are sometimes useful in comparing sequential and parallel algorithms. Suppose that we have a parallel algorithm that uses p processors. Let T_1

number of nodes	network density	number of slave processes				
		1	2	3	4	5
10	10%	0.50	0.50	1.00	1.00	1.00
	20%	1.00	1.00	2.00	2.00	2.00
	40%	0.86	1.17	1.17	1.17	1.75
	60%	0.86	1.00	1.00	1.20	1.20
	80%	0.88	1.17	1.40	1.40	1.40
20	10%	0.50	0.50	1.00	1.00	1.00
	20%	0.83	1.25	2.50	2.50	2.50
	40%	0.70	1.17	1.17	1.17	1.75
	60%	0.93	1.25	1.25	1.50	1.50
	80%	0.92	1.22	1.83	2.20	2.20
50	10%	0.91	1.25	1.67	2.00	2.00
	20%	1.00	1.42	1.89	2.43	2.43
	40%	0.95	1.33	2.00	2.50	2.86
	60%	0.91	1.33	1.82	2.22	2.50
	80%	0.85	1.34	1.70	2.44	2.75
100	10%	0.83	1.36	1.86	2.14	2.14
	20%	0.86	1.50	2.00	2.70	3.40
	40%	0.92	1.62	2.12	2.61	3.09
	60%	1.00	1.65	2.11	2.71	3.45
	80%	0.87	1.38	2.00	2.67	3.33
150	10%	0.83	1.13	1.40	1.84	2.06
	20%	1.00	1.43	1.85	2.38	2.94
	40%	0.93	1.27	1.73	2.08	2.12
	60%	0.94	1.31	2.00	2.28	2.56
	80%	0.95	1.32	1.94	2.26	2.41
200	10%	0.97	1.45	2.04	2.43	2.57
	20%	0.93	1.39	1.92	2.49	2.63
	40%	0.96	1.35	2.30	2.40	2.92
	60%	0.98	1.39	2.17	2.39	2.85
	80%	0.98	1.48	2.24	2.37	2.80

Table 1. Speedups obtained by the *PVM* implementation of our parallel algorithm

number of nodes	network density	number of slave processes				
		1	2	3	4	5
10	10%	0.02	0.02	0.01	0.01	0.01
	20%	0.02	0.02	0.01	0.01	0.01
	40%	0.04	0.03	0.03	0.03	0.02
	60%	0.07	0.06	0.06	0.05	0.05
	80%	0.08	0.06	0.05	0.05	0.05
20	10%	0.02	0.02	0.01	0.01	0.01
	20%	0.03	0.02	0.01	0.01	0.01
	40%	0.05	0.03	0.03	0.03	0.02
	60%	0.08	0.06	0.06	0.05	0.05
	80%	0.12	0.09	0.06	0.05	0.05
50	10%	0.11	0.08	0.06	0.05	0.05
	20%	0.17	0.12	0.09	0.07	0.07
	40%	0.21	0.15	0.10	0.08	0.07
	60%	0.22	0.15	0.11	0.09	0.08
	80%	0.26	0.16	0.13	0.09	0.08
100	10%	0.18	0.11	0.08	0.07	0.07
	20%	0.28	0.16	0.12	0.09	0.07
	40%	0.37	0.21	0.16	0.13	0.11
	60%	0.38	0.23	0.18	0.14	0.11
	80%	0.46	0.29	0.20	0.15	0.12
150	10%	0.42	0.31	0.25	0.19	0.17
	20%	0.50	0.35	0.27	0.21	0.17
	40%	0.56	0.41	0.30	0.25	0.24
	60%	0.68	0.49	0.32	0.28	0.25
	80%	0.74	0.53	0.36	0.31	0.29
200	10%	0.93	0.62	0.44	0.37	0.35
	20%	0.99	0.66	0.48	0.37	0.35
	40%	1.12	0.80	0.47	0.45	0.37
	60%	1.18	0.84	0.54	0.49	0.41
	80%	1.25	0.83	0.55	0.52	0.44

Table 2. Actual computing times in seconds

be the time required for the sequential algorithm and T_p the time required for the parallel algorithm using p processors. The ratio $\frac{T_1}{T_p}$ is called the speedup of the algorithm, and describes the speed advantage of the parallel algorithm, compared to the sequential algorithm. For a fixed number of nodes and fixed network density, Mathematica created 10 networks. The corresponding T_p was taken as the average computing time. Some of the obtained speedups are given in Table 1. Note that the average speedup ranges from 1.26 (using two slave processes) to 2.34 (using five processes). The maximal speedup obtained is 3.45. Actual computing times (in seconds) are given in Table 2.

7. Conclusion

Most of the parallel max-flow algorithms found in literature are based on expressing the original problem as a linear program, and on solving that linear program in parallel. Our algorithm uses a different approach: it directly solves the original network problem in parallel. The obtained experimental results clearly show that the used approach is successful. Our algorithm achieves a satisfactory speedup and can be recommended for networks which have more then 50 nodes and which are not too dense. Our future plan is

to make a more detailed evaluation with a large number of comparably fast processors.

References

J. M. CHRICHLOW, *An Introduction to Distributed and Parallel Programming*, Prentice Hall, Englewood Cliffs NJ, 1997.

A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, V. SUNDERAM, *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge MA, 1994.

A. V. GOLDBERG, A new max-flow algorithm , M.I.T., Technical Report MIT/LCS/ TM-291, Cambridge MA, 1985.

A. V. GOLDBERG, R. E.TARJAN, A new approach to the maximum flow problem, *Journal of ACM,* **35** (1988), 921-940.

B. KORTE, L. LOVÁSZ, H. J. PRÖMEL, A. SCHRIJVER, *Paths, Flows, and VLSI-Layout*, Springer-Verlag, New York, 1990.

G. NOGO, Parallel algorithms for network flow problems, PhD. Thesis (in Croatian), University of Zagreb, Zagreb, 1998.

L. PADBERG, T. Y. SUNG , An analytic symmetrization of max flow - min cut, *Discrete Mathematics,* **165/166** (1997), 531-545.

Y. SHILOACH, U. VISHKIN, An $O(n^2 \log n)$ parallel max-flow algorithm, *Journal of Algorithms,* **3** (1982), 128-146.

WOLFRAM RESEARCH., *Mathematica 3.0 Standard Add-on Packages*, Cambridge University Press, Cambridge UK, 1996.

Received: December, 1998
Accepted: April, 1999

Contact address:
Goranka Nogo
Department of Mathematics
University of Zagreb
Bijenicka 30
10000 Zagreb
Croatia
e-mail: nogo@math.hr

Robert Manger
Department of Mathematics
University of Zagreb
Bijenicka 30
10000 Zagreb
Croatia
e-mail: manger@math.hr

GORANKA NOGO received the BSc (1981), MSc (1985), and PhD (1998) degrees in mathematics, all from the University of Zagreb. She has been with the Department of Mathematics at the University of Zagreb since 1983 where she is presently a teaching assistant. During last years her work is centered around the theory of complexity, parallel algorithms and network flow problems. She has practical experience in programming and computing. Dr Nogo is a member of the Croatian Mathematical Society and the Mathematica Reference Center.

ROBERT MANGER received the BSc (1979), MSc (1982), and PhD (1990) degrees in mathematics, all from the University of Zagreb. For more than ten years he worked in industry, where he obtained practical experience in programming, computing, and designing information systems. Dr Manger is presently a lecturer in the Department of Mathematics at the University of Zagreb. His current research interests include parallel algorithms and neural networks. Dr Manger is a member of the Croatian Mathematical Society, Croatian Society for Operations Research and IEEE Computer Society.