Experimental Evaluation of a Parallel Max-Flow Algorithm

Goranka Nogo and Robert Manger

Department of Mathematics, University of Zagreb, Bijenička 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 $u : E \rightarrow \mathbb{R}$. 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 $f : E \rightarrow \mathbb{R}$ 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 anti-symmetry constraint).

We define the excess function $e_f : V \rightarrow \mathbb{R}$ 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 $u_f : E \rightarrow \mathbb{R}$ 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) ∈ E do begin
  f(v, w) ← 0;
  if v = s then f(v, w) ← u(v, w);
  if w = s then f(v, w) ← -u(w, v);
end;
for w ∈ V do begin
  e_f(w) ← \textstyle\sum_{(v,w)∈E} f(v, w);
  if w = s then d(w) ← n else d(w) ← 0;
end;
(*main loop*)
while ∃ 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) ∈ E then
  d(v) ← \textstyle\min_{(v,w)∈E'}\{d(w)\} + 1;
else if d(w) ≥ d(v), \forall(v, w) ∈ E then
  d(v) ← \textstyle\min_{(v,w)∈E'}\{d(w)\} + 1;
end.
```

```
procedure parallel_max_flow_algorithm(V, E, u)
(*initialisation*)
initialisation of $f$;
initialisation of $e_f$ and $d$;
initialisation of $L$;
```

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 ∪ list_of_predecessors,
```

is attached in fixed but arbitrary order.

The procedure $\text{pass_list}()$ is applicable to an active node $v$. It processes the $\text{list}(v)$. We have two passes through the $\text{list}(v)$ starting from the first node $w$ in the $\text{list}(v)$. If a pushing operation is applicable to the arc $(v, w)$, the procedure $\text{pass_list}$ pushes the excess of $v$ through the arc $(v, w)$. If not, $w$ is replaced with the next node in the $\text{list}(v)$ 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 $\mathcal{L}$ 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 $\mathcal{L} := \{l_1 \cup l_2 \cup \ldots \cup l_0 \}$ in fixed order (some lists may be empty). The procedure $\text{remove_node}(v)$ removes the first node $v$ from $\mathcal{L}$, and calls the procedure $\text{pass_list}$. Let $p$ be the number of processors, $|\mathcal{L}|$ the length of $\mathcal{L}$, and $\{v_1, \ldots, v_k\}$ the first $k$ elements (nodes) of $\mathcal{L}$, where $k = \text{min}\{p, |\mathcal{L}|\}$. 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 $\mathcal{L}$;
```

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 admissi-
(*main loop*)
while $\mathcal{L} \neq \emptyset$ do begin
    evaluation of $k$ and $\{v_1, ..., v_k\}$;
    (*main parallel loop*)
    for $j \in \{1, ..., k\}$ do in parallel
        remove_node($v_j$);
        parallel update of $e_f$ and $d$;
        parallel update of $\mathcal{L}$;
    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 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.

![Fig. 4. A sample network given as input to our algorithm](image-url)
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 $L$.

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 $T_1$

<table>
<thead>
<tr>
<th>number of nodes</th>
<th>network density</th>
<th>number of slave processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10%</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>20%</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>40%</td>
<td>0.86</td>
<td>1.17</td>
</tr>
<tr>
<td>60%</td>
<td>0.86</td>
<td>1.00</td>
</tr>
<tr>
<td>80%</td>
<td>0.88</td>
<td>1.17</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10%</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>20%</td>
<td>0.83</td>
<td>1.25</td>
</tr>
<tr>
<td>40%</td>
<td>0.70</td>
<td>1.17</td>
</tr>
<tr>
<td>60%</td>
<td>0.93</td>
<td>1.25</td>
</tr>
<tr>
<td>80%</td>
<td>0.92</td>
<td>1.22</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10%</td>
<td>0.91</td>
<td>1.25</td>
</tr>
<tr>
<td>20%</td>
<td>1.00</td>
<td>1.42</td>
</tr>
<tr>
<td>40%</td>
<td>0.95</td>
<td>1.33</td>
</tr>
<tr>
<td>60%</td>
<td>0.91</td>
<td>1.33</td>
</tr>
<tr>
<td>80%</td>
<td>0.85</td>
<td>1.34</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10%</td>
<td>0.83</td>
<td>1.36</td>
</tr>
<tr>
<td>20%</td>
<td>0.86</td>
<td>1.50</td>
</tr>
<tr>
<td>40%</td>
<td>0.92</td>
<td>1.62</td>
</tr>
<tr>
<td>60%</td>
<td>1.00</td>
<td>1.65</td>
</tr>
<tr>
<td>80%</td>
<td>0.87</td>
<td>1.38</td>
</tr>
<tr>
<td>150</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10%</td>
<td>0.83</td>
<td>1.13</td>
</tr>
<tr>
<td>20%</td>
<td>1.00</td>
<td>1.43</td>
</tr>
<tr>
<td>40%</td>
<td>0.93</td>
<td>1.27</td>
</tr>
<tr>
<td>60%</td>
<td>0.94</td>
<td>1.31</td>
</tr>
<tr>
<td>80%</td>
<td>0.95</td>
<td>1.32</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10%</td>
<td>0.97</td>
<td>1.45</td>
</tr>
<tr>
<td>20%</td>
<td>0.93</td>
<td>1.39</td>
</tr>
<tr>
<td>40%</td>
<td>0.96</td>
<td>1.35</td>
</tr>
<tr>
<td>60%</td>
<td>0.98</td>
<td>1.39</td>
</tr>
<tr>
<td>80%</td>
<td>0.98</td>
<td>1.48</td>
</tr>
</tbody>
</table>

Table 1. Speedups obtained by the PVM implementation of our parallel algorithm
<table>
<thead>
<tr>
<th>number of nodes</th>
<th>network density</th>
<th>number of slave processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10%</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>0.08</td>
</tr>
<tr>
<td>20</td>
<td>10%</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>0.12</td>
</tr>
<tr>
<td>50</td>
<td>10%</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>0.26</td>
</tr>
<tr>
<td>100</td>
<td>10%</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>0.46</td>
</tr>
<tr>
<td>150</td>
<td>10%</td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>0.74</td>
</tr>
<tr>
<td>200</td>
<td>10%</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>1.18</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>1.25</td>
</tr>
</tbody>
</table>

*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


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.