# Sequential and parallel deficit scaling algorithms for minimum flow in bipartite networks

LAURA CIUPALĂ
Department of Computer Science
University Transilvania of Braşov
Iuliu Maniu Street 50, Braşov
ROMANIA
laura_ciupala@yahoo.com

ELEONOR CIUREA
Department of Computer Science
University Transilvania of Braşov
Iuliu Maniu Street 50, Braşov
ROMANIA
e.ciurea@unitbv.ro

*Abstract:* - In this paper, first we describe the deficit scaling algorithm for minimum flow in bipartite networks. This algorithm is obtained from the deficit scaling algorithm for minimum flow in regular networks developed by Ciupală in [5] by replacing a pull from a node with sufficiently large deficit with two consecutive pulls. This replacement ensures that only nodes in $N_1$ can have deficits. Consequently, the running time of the deficit scaling algorithm for minimum flow is reduced from $O(nm+n^2 \log C)$ to $O(n_1 m+n_1^2 \log C)$ when it is applied on bipartite networks.

In the last part of this paper, we develop a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks on an EREW PRAM. The parallel bipartite deficit scaling algorithm performs a pull from an active node with a sufficiently large deficit and with the smallest distance label from $N_1$ at a time followed by a set of pulls from several nodes in $N_2$ in parallel. It runs in $O(n_1^2 \log C \log p)$ time on an EREW PRAM with $p = \lceil m/n_1 \rceil$ processors, which is within a logarithmic factor of the running time of the sequential bipartite deficit scaling algorithm for minimum flow.

*Key-Words:* - Network flow; Network algorithms; Bipartite network; Parallel algorithms; Minimum flow problem; Scaling technique

## 1 Introduction

The literature on network flow problem is extensive. Over the past 50 years researchers have made continuous improvements to algorithms for solving several classes of problems. From the late 1940s through the 1950s, researchers designed many of the fundamental algorithms for network flow, including methods for maximum flow and minimum cost flow problems. In the next decades, there are many research contributions concerning improving the computational complexity of network flow algorithms by using enhanced data structures, techniques of scaling the problem data etc.

Although it has its own applications, the minimum flow problem was not dealt so often as the maximum flow ([1], [2], [3], [14], [15], [16], [17], [18]) and the minimum cost flow problem ([1], [6], [20]).

There are many problems that occur in economy that can be reduced to minimum flow problems.

For instance, we present the machine setup problem. A job shop needs to perform $p$ tasks on a particular day. It is known the start time $\pi(i)$ and the end time $\pi'(i)$ for each task $i$, $i$=1,...,$p$. The workers must perform these tasks according to this schedule so that exactly one worker performs each task. A worker cannot work on two jobs at the same time. It is known the setup time $\pi_2(i, j)$ required for a worker to go from task $i$ to task $j$. We wish to find the minimum number of workers to perform the tasks.

We can formulate this problem as a minimum flow problem in the network $G = (N, A, l, c, s, t)$, determined in the following way:

$N = N_1 \cup N_2 \cup N_3 \cup N_4$,
$N_1 = \{s\}$,
$N_2 = \{i \mid i=1,...,p\}$,
$N_3 = \{i' \mid i'=1,...,p\}$,
$N_4 = \{t\}$,
$A = A_1 \cup A_2 \cup A_3 \cup A_4$,
$A_1 = \{(s, i) \mid i \in N_2\}$,
$A_2 = \{(i, i') \mid i, i'=1,...,p\}$,
$A_3 = \{(i', j) \mid \pi'(i')+\pi_2(i', j) \le \pi(j)\}$,
$A_4 = \{(i', t) \mid i' \in N_3\}$,
$l(s, i)=0$, $c(s, i)=1$, for any $(s, i) \in A_1$,
$l(i, i')=1$, $c(i, i')=1$, for any $(i, i') \in A_2$,
$l(i', j)=0$, $c(i', j)=1$, for any $(i', j) \in A_3$,
$l(i', t)=0$, $c(i', t)=1$, for any $(i', t) \in A_4$.

We solve the minimum flow problem in the network $G=(N, A, l, c, s, t)$ and the value of the minimum flow is the minimum number of workers that can perform the tasks.

The minimum flow problem in a network can be solved in two phases:

(1) establishing a feasible flow, if there is one

(2) from a given feasible flow, establish the minimum flow.

The first phase, i.e. the problem of determining a feasible flow, can be reduced to a maximum flow problem (for details see [1]).

For the second phase of the minimum flow problem there are three approaches:

1. using decreasing path algorithms (see [11], [12])
2. using preflow algorithms (see [5], [8], [9], [12])
3. using minimax algorithm which consists of finding a maximum flow from the sink node to the source node in the residual network (see [3], [10]).

The decreasing path algorithms work in the following way: they start with a feasible flow and they proceed by determining decreasing paths and by decreasing the flow along these paths. Any decreasing path algorithm terminates when the network contains no decreasing path, which means that the flow obtained is a minimum flow. The generic decreasing path algorithm for minimum flow does not specify any rule for determining the decreasing paths. By specifying different rules, many different algorithms were developed, which have better running times then the running time of the generic decreasing path algorithm.

The preflow algorithms for minimum flow begin with a feasible flow and send back as much flow, as it is possible, from the sink node to the nodes that are its neighbors, creating deficits in these nodes. The basic operation of any preflow algorithm for minimum flow is to select an active node (which is an intermediate node with a strictly negative deficit) and to send the flow entering in it back, closer to the source. For measuring closeness, distance labels are used. Any preflow algorithm for minimum flow terminates when the network contains no more active nodes, which means that the preflow is a flow. Moreover, it is a minimum flow. The generic preflow algorithm for minimum flow does not specify any rule for selecting active nodes. By specifying different rules we can develop many different algorithms, which can have better running times then the generic preflow algorithm. The deficit scaling algorithm, developed by Ciupală in [5], always selects an active node with a sufficiently large deficit.

A third approach of the minimum flow problem consists of determining a maximum flow from the sink node to the source node in the residual network. For this, any maximum flow algorithm can be used.

The algorithms in each of these three classes (decreasing path algorithms, preflow algorithms, minimax algorithm) can be modified in order to become more efficient when they are applied on bipartite networks.

In this paper, first we describe the deficit scaling algorithm for minimum flow in bipartite networks. This algorithm is obtain from the deficit scaling algorithm for minimum flow in regular networks developed by Ciupală in [5] by replacing a pull from a node with deficit with two consecutive pulls. This replacement ensures that only nodes in $N_1$ can have deficits. Consequently, the running time of the deficit scaling algorithm is reduced from $O(nm + n^2 \log C)$, which is the running time of the deficit scaling algorithm applied on regular networks, to $O(n_1 m + n_1^2 \log C)$.

In section 5, we develop a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks. This algorithm performs a pull from an active node with a large deficit and with the smallest distance label from $N_1$ at a time followed by a set of pulls from several nodes in $N_2$ in parallel. On a PRAM with $p = \lceil m/n_1 \rceil$ processors, it runs in $O(n_1^2 \log C \log p)$ time.

## 2 Notation and definition

We consider a capacitated network $G = (N, A, l, c, s, t)$ with a nonnegative capacity $c(i, j)$ and with a nonnegative lower bound $l(i, j)$ associated with each arc $(i, j) \in A$. We distinguish two special nodes in the network $G$: a source node $s$ and a sink node $t$.

Let $n=|N|$, $m = |A|$ and $C = \max \{ c(i, j) \mid (i, j) \in A \}$.

A *flow* is a function $f : A \rightarrow \mathbf{R}_+$ satisfying the next conditions:

$$f(s, N) - f(N, s) = v \quad (1)$$

$$f(i, N) - f(N, i) = 0, \; i \neq s,t \quad (2)$$

$$f(t, N) - f(N, t) = -v \quad (3)$$

$$l(i, j) \leq f(i, j) \leq c(i, j), \; (i, j) \in A \quad (4)$$

for some $v \geq 0$, where

$$f(i, N) = \Sigma_j f(i, j), \; i \in N$$

and

$$f(N, i) = \Sigma_j f(j, i), \; i \in N.$$

We refer to $v$ as the *value* of the flow $f$.

The minimum flow problem is to determine a flow $f$ for which $v$ is minimized.

For the minimum flow problem, a *preflow* is a function $f : A \rightarrow \mathbf{R}_+$ satisfying the next conditions:

$$f(i, N) - f(N, i) \leq 0, \; i \neq s,t \quad (5)$$

$$l(i, j) \leq f(i, j) \leq c(i, j), \; (i, j) \in A \quad (6)$$

Let $f$ be a preflow. We define the *deficit* of a node $i \in N$ in the following manner:

$$e(i) = f(i, N) - f(N, i) \quad (7)$$

Thus, for the minimum flow problem, for any preflow $f$, we have $e(i) \leq 0$, $i \in N \setminus \{s, t\}$.

We say that a node $i \in N \setminus \{s, t\}$ is *active* if $e(i) < 0$ and *balanced* if $e(i) = 0$.

A preflow $f$ for which

$$e(i) = 0, \; i \in N \setminus \{s, t\}$$

is a flow. Consequently, a flow is a particular case of preflow.

For the minimum flow problem, the *residual capacity* $r(i, j)$ of any arc $(i, j) \in A$, with respect to a given preflow $f$, is given by

$$r(i, j) = c(j, i) - f(j, i) + f(i, j) - l(i, j).$$

By convention, if $(i, j) \in A$ and $(j, i) \notin A$, then we add the arc $(j, i)$ to the set of arcs $A$ and we set $l(j, i) = 0$ and $c(j, i) = 0$. The residual capacity $r(i, j)$ of the arc $(i, j)$ represents the maximum amount of flow from the node $i$ to node $j$ that can be canceled by modifying the flow on both of the arcs $(i, j)$ and $(j, i)$.

The network $G_f = (N, A_f)$ consisting only of those arcs with strictly positive residual capacity is referred to as the *residual network* (with respect to the given preflow $f$).

In the residual network $G_f = (N, A_f)$ the *distance function* $d : N \rightarrow \mathbf{N}$ with respect to a given preflow $f$ is a function from the set of nodes to the nonnegative integers.

We say that a distance function is *valid* if it satisfies the following validity conditions:

$d(s) = 0$

$d(j) \le d(i) + 1$, for every arc $(i, j) \in A_f$.

We refer to $d(i)$ as the distance label of node $i$.

**Theorem 1.** *(a) If the distance labels are valid, the distance label $d(i)$ is a lower bound on the length of the shortest directed path from node $s$ to node $i$ in the residual network.*

*(b) If $d(t) \ge n$, the residual network contains no directed path from the source node $s$ to the sink node $t$.*

*Proof.* (a) Let $P = (s=i_1, i_2, \ldots, i_k, i_{k+1}=i)$ be any path of length $k$ from node $s$ to node $i$ in the residual network. The validity conditions imply that:

$$d(i_2) \le d(i_1)+1 = d(s)+1 = 1$$
$$d(i_3) \le d(i_2)+1 \le 2$$
$$d(i_4) \le d(i_3)+1 \le 3$$
$$\ldots.$$
$$d(i_{k+1}) \le d(i_k)+1 \le k.$$

(b) We proved that $d(t)$ is a lower bound on the length of the shortest path from the source node $s$ to the sink node $t$ in the residual network and we know that no directed path can contain more than $(n-1)$ arcs. Consequently, if $d(t) \ge n$, then the residual network contains no directed path from $s$ to $t$.

We say that the distance labels are *exact* if for each node $i$, $d(i)$ equals the length of the shortest path from node $s$ to node $i$ in the residual network.

We refer to an arc $(i, j)$ from the residual network as an *admissible arc* if $d(j) = d(i) + 1$; otherwise it is *inadmissible*.

We refer to a node $i$ with $e(i) < 0$ as an *active* node. We adopt the convention that the source node and the sink node are never active.

# 3 Deficit scaling algorithm

This algorithm is a special implementation of the generic preflow algorithm for minimum flow. This algorithm was developed by Ciurea and Ciupală in [12] and it begins with a feasible flow and sends back as much flow, as it is possible, from the sink node to the source node. Because the algorithm decreases the flow on individual arcs, it does not satisfy the mass balance constraint (1), (2), (3) at intermediate stages. In fact, it is possible that the flow entering in a node exceeds the flow leaving from it. Such a node is an active node because is has a strictly negative deficit. The basic operation of this algorithm is to select an active node and to send the flow entering in it back, closer to the source. For measuring closeness, the generic preflow algorithm for minimum flow uses the distance labels $d(\cdot)$. Suppose that $j$ is a node with strictly negative deficit selected by the algorithm. If it exists an admissible arc $(i, j)$, it pulls flow on this arc; otherwise it relabels the node $j$ in order to create at least one admissible arc entering in the node $j$. The generic preflow algorithm for minimum flow repeats this process until the network contains no more active nodes, which means that the preflow is actually a flow. Moreover, it is a minimum flow.

The generic preflow algorithm for minimum flow does not specify any rule for selecting active nodes. By specifying different rules we can develop many different algorithms, which can have better running times then the generic preflow algorithm. For example, we could select active nodes in FIFO order, or we could always select the active node with the greatest distance label, or the active node with the minimum distance label, or the active node selected most recently or least recently, or the active node with the largest deficit or we could select any of active nodes with a sufficiently large deficit.

The deficit scaling algorithm, developed by Ciupală in [5], always selects an active node with a sufficiently large deficit. Like all preflow algorithms for minimum flow, the deficit scaling algorithm maintains a preflow at every step and proceeds by pulling the deficits of the active nodes closer to the source node. For measuring closeness it uses the exact distance labels. Consequently, pulling the deficits from the active nodes closer to the source node means decreasing flow on admissible arcs.

Let $e_{max} = \max \{-e(i) \mid i \text{ is an active node}\}$.

The *deficit dominator* is the smaller integer $\overline{r}$ that is a power of 2 and satisfies $e_{max} \le \overline{r}$. We refer to a node $i$ with $e(i) \le -\overline{r}/2$ as a node with *large deficit* and as a node with *small deficit* otherwise.

The scaling deficit algorithm for the minimum flow always pulls flow from active nodes with sufficiently large deficits to nodes with sufficiently small deficits in order to not allow that a deficit becomes too large. We try to avoid that a deficit of a node becomes too large because it is unlikely to succeed to send such a large amount of flow in totality back to the source node *s*. And the deficit that cannot be moved closer to the source node will be returned to the sink node, operations which imply some additional computations.

The deficit scaling algorithm for the minimum flow is the following:

**Deficit scaling algorithm;**
**begin**
let $f$ be a feasible flow in network $G$;
compute the exact distance labels $d(\cdot)$ in the residual network $G_f$;
**if** $t$ is not labeled **then**
$f$ is a minimum flow
**else**
**begin**
**for** each arc $(i, t) \in$ A **do**
$f(i, t) := l(i, t)$;
$d(t) := n$;
$\overline{r} := 2^{\lceil \log C \rceil}$;
**while** $\overline{r} \geq 1$ **do**
**begin**
**while** the network contains an active node with a large deficit **do**
**begin**
among active nodes with large deficits, select a node $j$ with the smallest distance label;
*pull_relabel*($j$);
**end**
$\overline{r} := \overline{r}$ /2;
**end**
**end**
**end.**

**Procedure pull_relabel(*j*)**
**begin**
**if** the network contains an admissible arc $(i, j)$ **then**
**if** $i \neq t$ **then**
pull g = min $\{-e(j), r(i, j), \overline{r} + e(i)\}$ units of flow from node $j$ to node $i$;
**else**
pull g = min $\{-e(j), r(i, j)\}$ units of flow from node $j$ to node $i$;
**else**
$d(j) := \min\{ d(i) \mid (i, j) \in A_f \} + 1$;
**end;**

Let us refer to a phase of the algorithm during which $\overline{r}$ remains constant as a *scaling phase* and a scaling phase with a specific value of $\overline{r}$ as a $\overline{r}$ *-scaling phase*.

**Theorem 2.** *If there exists a feasible flow in the network $G = (N, A, l, c, s, t)$, then the deficit scaling algorithm determines a minimum flow.*

*Proof*. The algorithm starts with $\overline{r} := 2^{\lceil \log C \rceil}$, $C \leq \overline{r} \leq 2C$. During the $\overline{r}$ -scaling phase, $e_{\max}$ might increase or decrease but it must meet the condition:

$$\overline{r}/2 < e_{\max} \leq \overline{r}.$$

When $e_{\max} \leq \overline{r}/2$ the algorithm halves the value of $\overline{r}$ and begins a new scaling phase. After $1+\lfloor \log C \rfloor$ scaling phases, $e_{\max}$ becomes 0 and we obtain a minimum flow.

Actually, the algorithm terminates with optimal residual capacities. From these residual capacities we can determine a minimum flow in several ways. For example, we can make a variable change: for all arcs $(i, j)$, let

$$c'(i, j) = c(i, j) - l(i, j),$$
$$r'(i, j) = r(i, j),$$
$$f'(i, j) = f(i, j) - l(i, j).$$

The residual capacity of arc $(i, j)$ is

$$r(i, j) = c(j, i) - f(j, i) + f(i, j) - l(i, j)$$

Equivalently,

$$r'(i, j) = c'(j, i) - f'(j, i) + f'(i, j).$$

We can compute the value of $f'$ in the following way:

$$f'(i, j) = \max(r'(i, j) - c'(j, i), 0).$$

Converting back into the original variables, we obtain the following expression:

$$f(i, j) = l(i, j) + \max(r(i, j) - c(j, i) + l(j, i), 0).$$

**Theorem 3.** During each $\overline{r}$ -scaling phase, the algorithm satisfies the following two conditions:

(a) each noncanceling pull decreases the flow by at least $\overline{r}/2$ units

(b) $e_{\max} \leq \overline{r}$.

*Proof*. (a) We consider a noncanceling pull on arc $(i, j)$. Since $(i, j)$ is an admissible arc, $d(j) = d(i) + 1 > d(i)$. But, $j$ is a node with a smallest distance label among all nodes with a large deficit. Thus, $e(j) \leq -\overline{r}/2$ and $e(i) > -\overline{r}/2$. Since this pull is a noncanceling pull, it decreases the flow by $\min\{-e(j), \overline{r} + e(i)\} \geq \overline{r}/2$.

(b) A pull on arc $(i, j)$ increases only the absolute value of the deficit of node $i$. The new deficit of node $i$ is $e'(i) = e(i) - \min\{-e(j), r(i, j), \overline{r} + e(i)\} \geq e(i) - (\overline{r} + e(i)) = -\overline{r}$. Thus, $e'(i) \geq -\overline{r}$ and $e_{\max} \leq \overline{r}$.

**Theorem 4.** For each node $i \in N$, $d(i) < 2n$.

This theorem can be proved in a manner similar to the

proof of the corresponding theorem from the complexity analysis of the generic preflow algorithm (for details see [12]).

**Theorem 5.** During each scaling phase, the algorithm performs $O(n^2)$ noncanceling pulls.

*Proof*. We consider the potential function $F = -\sum e(i)d(i)/\overline{r}$ . The initial value of $F$ at the beginning of the $\overline{r}$ -scaling phase is bounded by $2n^2$ because $e(i) \geq -\overline{r}$ and $d(i) \leq 2n$ for all $i \in N$ (from Theorem 3 and Theorem 4).

After the algorithm has selected node $j$, one of the following two cases must apply:

Case 1. The algorithm is unable to find an admissible arc along which it can pull flow. In this case, the distance label of node $j$ increases by $q \geq 1$ units. This increases $F$ by at most $q$ units because $e(i) \geq -\overline{r}$ . Since for each node $i$ the total increase in $d(i)$ throughout the running of the algorithm is bounded by $2n$ (from Theorem 4), the total increase in $F$ due to the relabelings of nodes is bounded by $2n^2$.

Case 2. The algorithm is able to find an admissible arc along which it can pull flow, so it performs either a cancelling or a noncancelling pull. In either case, $F$ decreases. After a noncancelling pull on arc $(i, j)$, the flow from node $i$ to node $j$ decreases by at least $\overline{r}/2$ units and $F$ decreases by at least 1/2 units because $d(j) = d(i) + 1$. As the initial value of $F$ at the beginning of the scaling phase plus the increase in $F$ sum to at most $4n^2$, this case cannot occur more than $8n^2$ times. Thus, the algorithm performs $O(n^2)$ noncanceling pulls per scaling phase.

**Theorem 6.** The deficit scaling algorithm runs in $O(nm + n^2 \log C)$ time.

*Proof.* Since the algorithm performs $O(\log C)$ scaling phase, from Theorem 5 it follows that the algorithm performs $O(n^2 \log C)$ noncanceling pulls in total. The other operations (cancelling pulls, relabel operations and finding admissible arcs) require $O(nm)$ time (this can be proved in a similar way as Ciurea and Ciupală proved the complexity of the generic preflow algorithm in [11]). Consequently, the deficit scaling algorithm runs in $O(nm + n^2 \log C)$ time.

## 4 Deficit scaling algorithm for minimum flow in bipartite networks

A network $G = (N, A)$ is called *bipartite* if its node set $N$ can be partitioned into two subsets $N_1$ and $N_2$, such that all arcs have one endpoint in $N_1$ and the other in $N_2$.

Let $n_1 = |N_1|$, $n_2 = |N_2|$. We assume, without loss of generality, $n_1 \leq n_2$.

We consider a bipartite capacitated network $G = (N_1, N_2, A, l, c, s, t)$ with a nonnegative capacity $c(i, j)$ and with a nonnegative lower bound $l(i, j)$ associated with each arc $(i, j) \in A$. We distinguish two special nodes in the network $G$: a source node $s$ and a sink node $t$. We assume, without loss of generality, that $s \in N_1$ and $t \in N_2$.

The basic idea behind the deficit scaling algorithm for minimum flow in a bipartite network is to perform bipulls from nodes in $N_1$. A *bipull* is a pull over two consecutive admissible arcs. Consequently, a bipull moves the deficit from a node in $N_1$ to another node in $N_1$. This approach has all the advantages of the scaling deficit algorithm for minimum flow in regular networks. Moreover, it has an additional advantage that leads to an improved running time. This additional advantage consists of the fact that, using bipulls instead of pulls, all the nodes in $N_2$ are maintained balanced.

We refer to a bipull along the path $h - i - j$ as *cancelling* if after it at least one of the arcs $(h, i)$ and $(i, j)$ is dropped from the residual network; otherwise the bipull is *noncancelling*.

Obviously, after a noncancelling bipull along the path $h - i - j$, the deficit of the node $j$ becomes 0.

Since all the deficits are at the nodes in $N_1$, it is sufficient to account for the noncancelling bipulls from the nodes in $N_1$. Since $|N_1| < |N|$, the number of noncancelling bipulls is reduced.

For determining a minimum flow in a bipartite network, we can use the deficit scaling algorithm modified by replacing the procedure *pull_relabel* with the procedure *bipull_relabel*, which is described below.

**Procedure bipull_relabel(*j*)**
**begin**
**if** the network contains an admissible arc $(i, j)$ **then**
  **if** the network contains an admissible arc $(h, i)$ **then**
    **if** $h \neq t$ **then**
      pull g = min $\{-e(j), r(i, j), r(h, i), \overline{r} + e(h)\}$ units of flow along the path $h–i–j$ ;
    **else**
      pull g = min $\{-e(j), r(i, j)\ r(h, i)\}$ units of flow along the path $h –i - j$ ;
  **else** $d(i) := \min\{ d(h) \mid (h, i) \in A_f \} + 1$;
**else** $d(j) := \min\{ d(i) \mid (i, j) \in A_f \} + 1$;
**end;**

**Theorem 7.** *If there exists a feasible flow in the bipartite network $G = (N, A, l, c, s, t)$, then the deficit scaling algorithm for minimum flow in a bipartite network determines a minimum flow.*

*Proof*. The proof of this theorem follows directly from the Theorem 2.

**Theorem 8.** During the execution of the deficit scaling algorithm for minimum flow in a bipartite network, all the deficits remain on the nodes in $N_1$.

*Proof.* At the beginning of the algorithm, one pulls as much flow as it is possible on the arcs entering in the sink node $t \in N_2$. This operation creates deficits in the nodes that are in the neighborhood of the sink node $t$. Thus, all the nodes with deficit after initializations are in $N_1$. All the other pulls of flow in the algorithm are done using the procedure *bipull_relabel*, which pulls flow from a node in $N_1$ through a node in $N_2$ to another node in $N_1$, never leaving any deficit on a node in $N_2$. No other operations create deficit at any node.

Using the result from Theorem 8, we can prove in the same manner as we proved Theorem 5 that during each scaling phase, the deficit scaling algorithm for minimum flow in a bipartite network performs $O(n_1^2)$ noncancelling pulls.

From Theorem 8 it follows that the other operations performed by the deficit scaling algorithm for minimum flow in a bipartite network (cancelling pulls, relabel operations and finding admissible arcs) can be done in $O(n_1 m)$ time. Consequently, we obtained the following result:

**Theorem 9.** The deficit scaling algorithm for minimum flow in a bipartite network, runs in $O(n_1 m + n_1^2 \log C)$ time.

# 5 Parallel deficit scaling algorithm for minimum flow in bipartite networks

In this section, we develop a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks, developed in section 4, on an EREW PRAM using $p = \lceil m/n_1 \rceil$ processors. This algorithm can be applied on networks in which any node has both in-degree and out-degree no greater than $p$. This restriction implies no loss of generality because any bipartite network with $m$ arcs, $n_1$ nodes in $N_1$ and $n_2$ nodes in $N_2$ can be transformed in an equivalent bipartite network with $O(m)$ arcs, $O(n_1)$ nodes in $N_1$ and $O(n_2)$ nodes in $N_2$ in which any node has both in-degree and out-degree no greater than $p$.

First we determine a transformed network in which all the nodes have out-degree no greater then $p$ and that is equivalent to the original network. This network is determined from the original network in the following manner:

select a node $j$ with out-degree $k > p$; we denote the arcs outgoing from the node $j$ by $(j, j_1), (j, j_2),..., (j, j_k)$.

create two new nodes $j'$ and $j''$,

replace the arcs the arcs $(j, j_{k-p+1}), ..., (j, j_k)$ with the arcs $(j, j')$, $(j', j'')$ and $(j'', j_{k-p+1}), ..., (j'', j_k)$,

set $l(j, j') = 0$, $c(j, j') = \infty$, $l(j', j'') = 0$, $c(j', j'') = \infty$

set $l(j'', j_h) = l(j, j_h)$ and $c(j'', j_h) = c(j, j_h)$ for each $h$ from $k - p + 1$ to $k$

repeat this process until the network contains no node with out-degree greater than $p$.

After this transformation, we obtained an equivalent network in which any node has out-degree no greater than $p$.

Each time a a node $j$ with out-degree greater than $p$ is selected, a new node is added to $N_1$, a new node is added to $N_2$ and two more arcs are added to $A$.

In the same manner, we can determine a transformed network in which all the nodes have out-degree no greater then $p$ and that is equivalent to the original network.

We can transform any bipartite network with $m$ arcs, $n_1$ nodes in $N_1$ and $n_2$ nodes in $N_2$ can be transformed in an equivalent bipartite network with $O(m)$ arcs, $O(n_1)$ nodes in $N_1$ and $O(n_2)$ nodes in $N_2$ in which any node has both the out-degree no greater than $p$ in $O(n_1 \log m)$ time using $p$ processors (for details see [2]).

Consequently, for the rest of this section, we will assume, without loss of generality, that each node in the network has both in-degree and out-degree less than or equal to $p$.

For any node $j \in N$, let $N(j) = \{ i \in N \mid (i, j) \in A_f \}$. We assume that nodes in $N(j)$ are denoted by $j_1, j_2, ..., j_k$, where $k = |N(j)|$. Let $N'(j) = \{ i \in N \mid (i, j) \in A_f$ and $(i, j)$ is an admissible arc$\}$.

For each node $j \in N_2$, we refer to

$$r'(j) = \Sigma_{i \in N'(j)} r(i,j)$$

as the *effective residual capacity* of node $j$.

Note that we can always pull the entire deficit of a node $j$ before relabeling it as long as the deficit does not exceed the effective residual capacity of the node $j$.

We define the *effective residual capacity* $r'(i, j)$ of arc $(i, j)$ in the following manner:

$r'(i, j) = 0$ if $(i, j)$ is not an admissible arc

$r'(i, j) = r(i, j)$ if $(i, j)$ is an admissible arc and $i \in N_2$ and $j \in N_1$

$r'(i, j) = \min\{r(i, j), r'(i)\}$ if $(i, j)$ is an admissible arc and $i \in N_2$ and $j \in N_1$

The parallel deficit scaling algorithm for minimum flow in bipartite networks pulls flow from a node $j \in N_1$ with sufficiently large deficit at a time and then it pulls flow from several nodes from $N_2$ in parallel. Pulling $r'(i, j)$ units of flow on any arc $(i, j)$ with $i \in N_2$ and $j \in N_1$, we can be sure that we never pull more flow into a node

$i \in N_2$ than its effective residual capacity. Consequently, all the deficit of node $i$ can be pulled out prior to a relabeling of node $i$.

For an efficient allocation of the processors to the arcs (because we cannot assign one processor to each arc), we will use the following four functions:

*Current*($j$) will return the current arc entering into $j$

*NextCurrent*($j$, $g$) will return $|N(j)|+1$ if after pulling $g$ units of flow from node $j$ all admissible arc entering in $j$ will be dropped from the residual network. Otherwise, it will return the index of the arc that will become current arc after pulling $g$ units of flow from node $j$.

*NextDecrement*($j$, $g$) will return the amount of flow that will be pull on arc *NextCurrent*($j$, $g$) when pulling flow from node $j$.

*Allocate*($j$, $D$) takes as an input a node $j$ and a $p$-dimensional array of demands of processors from the nodes in $N(j)$ and returns a vector *proc*, where *proc*($k$) is the set of processors allocated to the node $j_k$ from $N(j)$.

Using these four functions described above, we can describe the parallel bipull_relabel procedure within the parallel bipartite deficit scaling algorithm for minimum flow in a bipartite network. This procedure performs at a time a pull from an active node with a large deficit and with the smallest distance label from $N_1$ followed by a set of parallel pulls from several active nodes in $N_2$, each of which is preceded by processor allocation. The parallel bipull_relabel procedure concludes by relabeling the necessary nodes.

The parallel deficit scaling algorithm for minimum flow in a bipartite network is the following:

**Parallel bipartite deficit scaling algorithm;**
**begin**
let $f$ be a feasible flow in network $G$;
compute the exact distance labels $d(\cdot)$ in the residual network $G_f$ by applying the BFS parallel algorithm from the source node $s$;
**if** $t$ is not labeled **then**
$f$ is a minimum flow
**else begin**
**for** each arc $(i, t) \in$ A **do in parallel**
$f(i, t) := l(i, t)$;
$d(t) := n$;
$\bar{r} := 2^{\lceil \log C \rceil}$;
**while** $\bar{r} \geq 1$ **do**
**begin**
**while** the network contains an active node with a large deficit **do**
**begin**
determine in parallel $d(j) = \min\{d(i) \mid i$ is an active node with large deficit$\}$;
*parallel bipull_relabel*($j$, $\bar{r}$ /2, *proc*($j$));
**end**
$\bar{r} := \bar{r}$ /2;
**end**
**end**
**end.**

**Procedure parallel bipull_relabel(*j, g, P*)**
**begin**
*parallel pull*($j$, $g$, $P$);
**while** $e(j_k) < 0$ for some $j_k \in N(j)$ **do**
**begin**
**for** $k = 1$ **to** $p$ **do in parallel**
$D(j_k) = NextCurrent(j_k, -e(j_k)) - Current(j_k)+1$;
*proc* = *Allocate*($j$, $D$);
**for** $k = 1$ **to** $p$ **do in parallel**
**begin**
*parallel pull*($j_k$, $-e(j_k)$, *proc*($k$));
update data structures;
**end;**
**for** each $j_k \in N(j)$ **do**
**if** $Current(j_k) = |N(j_k)| + 1$ **then**
relabel the node $j_k$;
**if** $Current(j) = |N(j)| + 1$ **then**
relabel the node $j$;
**end;**
**end;**

**Procedure parallel pull(*j, g, P*)**
**begin**
$c = Current(j)$;
$k = NextCurrent(j, g)$;
$s = |P|$;
**for** $i = c$ **to** $\min(k\text{-}1, c+s\text{-}1)$ **do in parallel**
pull $r'(j_i, j)$ units of flow on arc $(j_i, j)$ and update $r'$;
**if** $s \geq k\text{-}c+1$ **and** $k \leq |N(j)|$ **then**
pull *NextDecrement*($j$, $g$) units of flow on arc $(j_k, j)$ and update $r'$;
$Current(j) = NextCurrent(j, g)$;
**end;**

**Theorem 9.** *If there exists a feasible flow in the network $G = (N, A, l, c, s, t)$, then the parallel deficit scaling algorithm for minimum flow in a bipartite network determines a minimum flow.*

*Proof.* The proof of this theorem follows directly from the Theorem 7.

In order to determine the running time of the parallel bipartite deficit scaling algorithm for minimum flow in a bipartite network, first we must determine the running

time of each of the four functions: *Current*($j$), *NextCurrent*($j$, $g$), *NextDecrement*($j$, $g$) and *Allocate*($j$, $D$).

Assuming that $|N(j)|$ is a power of 2, we can associate to any node $j$ a complete binary tree $T(j)$ whose leaves are the indexes of the nodes in $N(j)$. The key of the leaf $k$ is $r'(j_k, j)$ and the key of each internal node of the binary tree is the sum of the keys of its descendent leaves.

When a node $j$ is relabeled, to each node $j_k$ of $N(j)$ is assigned a processor and its binary tree is updated. This assignment of processors takes $O(\log p)$ steps per relabel. Moreover, each processor updates its binary tree in $O(\log p)$ steps.

When a pull of flow from a node $j$ is performed, the binary tree for the node $j$ must be updated. If $k$ processors are assigned, the *Current*($j$) is increased by at most $k$ and the updating can be accomplished with $k$ processors in $O(\log p)$ time.

In order to compute *NextCurrent*($j$, $g$), we start at the root of the binary tree corresponding to the node $j$ and we select the right child or the left child depending on whether $g$ is less than or greater than the key of the right child. We then recur on the selected child. We also can compute *NextDecrement*($j$, $g$) in this manner. Obviously, both functions *NextCurrent*($j$, $g$) and *NextDecrement*($j$, $g$) can be computed using one processor in $O(\log p)$ time.

The function *Allocate*($j$, $D$) can be straightforward implemented using with prefix operations using $p$ processors in $O(\log p)$ time.

**Theorem 10.** *The parallel pull procedure runs in* $O(\log p)$ *time on* $p = \lceil m/n_1 \rceil$ *processors.*

*Proof.* The for loop can be implemented by a parallel prefix operation on $p$ processors. All the other steps can be implemented in $O(\log p)$ time using a single processor.

**Theorem 11.** *There are* $O(n_1^2 \log C)$ *calls to parallel bipull_relabel procedure over the course of the parallel bipartite deficit scaling algorithm.*

*Proof.* Each parallel bipull_relabel procedure in the first line either moves $\overline{r}/2$ units of flow or results in a relabeling. By a proof similar to that of the Theorem 5, there are at most $O(n_1^2 \log C)$ such pulls over the whole algorithm.

**Theorem 12.** *The parallel bipull_relabel procedure runs in* $O($*the number of iterations of the while loop* $\times \log p)$ *time using* $p = \lceil m/n_1 \rceil$ *processors.*

*Proof.* Each step except the parallel push procedure takes $O(\log p)$ time. From Theorem 10, we know that a parallel pull procedure takes $O(\log p)$ time. Obviously, a set of pulls which use a total of $p$ arcs can also be implemented in $O(\log p)$ time. Consequently, each iteration of the while loop can be implemented in $O(\log p)$ time.

**Theorem 13.** *The while loop in the parallel bipull_relabel procedure is executed* $O(n_1 m/p + n_1^2 \log C)$ *time over the course of the parallel bipartite deficit scaling algorithm.*

*Proof.* First, we note that each node in $N(j)$ may have at most one noncancelling pull over the whole execution of the while loop. From the Theorem 11, it follows that the number of noncancelling pulls is at most $O(n_1^2 p \log C)$ overall. Let $x$ be the number of the noncancelling pulls that were executed since the beginning of the algorithm. We consider the potential function $F = \Sigma_j Current(j) + x$. Initially, $F = 0$ and at termination $F =$ the number of nocancelling pulls $= O(n_1^2 p \log C)$. The only way for $F$ to decrease is by relabeling. Each relabeling of a node $j$ decreases $F$ by $|N(j)|$. Consequently, the total decrease of $F$, due to relabelings, is $O(n_1 m)$. Thus, the total increase in $F$ over the whole algorithm is $O(n_1^2 p \log C + n_1 m)$. A parallel pull using $k$ processors increases $F$ by $k$ or results in a relabeling. Ignoring the last iteration of the while loop in each parallel bipull_relabel, we find that there are at most $O((n_1^2 p \log C + n_1 m)/p)$ iterations of the while loop. Because there are at most $O(n_1^2 \log C)$ last iterations, overall there are $O(n_1^2 \log C + n_1 m/p)$ iterations.

Combining all the above results, we obtain the following theorem:

**Theorem 14** *The parallel bipartite deficit scaling algorithm determines a minimum flow in a bipartite network in* $O(n_1^2 \log C \log p)$ *time using* $p = \lceil m/n_1 \rceil$ *processors* .

Consequently, the parallel bipartite deficit scaling algorithm runs within a logarithmic factor of the running time of the sequential bipartite deficit scaling algorithm for minimum flow.

# 5 Conclusion

In this paper, we discussed the minimum flow problem, which is a network problem that is not often treated, although it has its own applications.

First, we described the deficit scaling algorithm for minimum flow in a bipartite network $G = (N_1, N_2, A, l, c, s, t)$. This algorithm was obtained from the deficit scaling algorithm for minimum flow in

regular networks developed by Ciupală in [5] by replacing a pull from a node with sufficiently large deficit with two consecutive pulls. This replacement ensured that only nodes in $N_1$ can have deficits and it implied a reduction of the running time of the deficit scaling algorithm for minimum flow from $O(nm + n^2 \log C)$ to $O(n_1 m + n_1^2 \log C)$ when it is applied on bipartite networks.

Then, we developed a parallel implementation of the deficit scaling algorithm for minimum flow in bipartite networks. This algorithm performs a pull from an active node with a large deficit and with the smallest distance label from $N_1$ at a time, followed by pulls from several nodes in $N_2$ in parallel. Consequently, it runs in $O(n_1^2 \log C \log p)$ time on a EREW PRAM with $p = \lceil m/n_1 \rceil$ processors, which is within a logarithmic factor of the running time of the sequential bipartite deficit scaling algorithm for minimum flow.

## 6 Further improvements

One needs to solve the problems from the real life more and more quickly. Solving some of these problems means solving network flow problems. This is one of the reasons for which the speeding-up of the network flow algorithms is researched. For obtaining more quickly a solution of a network flow problem, there are several approaches: designing more efficient algorithms, improving the running time of some of the existing algorithms (by using enhanced data structures or by using certain techniques, for instance technique of scaling the problem data etc.), developing parallel implementations of some of the existing algorithms etc. In this paper, we used two approaches: we improved a running time of the deficit scaling algorithm for minimum flow by using the particularities of the network on which it is applied (i.e., the particularities of a bipartite network) and we implemented it in parallel on an EREW PRAM.

The deficit scaling algorithm for minimum flow in bipartite networks, developed in this paper, was obtain from deficit scaling algorithm for minimum flow in regular networks by replacing a pull from a node with sufficiently large deficit with two consecutive pulls. The same transformation can be applied to any of the variants of the deficit scaling algorithm for minimum flow, thereby improving their running time when they are applied on bipartite networks.

*References:*

[1] R. Ahuja, T. Magnanti and J. Orlin, *Network flows. Theory, algorithms and applications*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1993.

[2] R. Ahuja, J. Orlin, C. Stein and R. Tarjan, Improved algorithms for bipartite network flow, *SIAM Journal of Computing* Vol. 23, 1994, pp. 906-933.

[3] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer-Verlag, London, 2001.

[4] J. Barros, S.D. Servetto, *Network Information Flow with Correlated Sources*, IEEE Transactions on Information Theory, 52(1), 155-170, 2006.

[5] L. Ciupală, A deficit scaling algorithm for the minimum flow problem, *Sadhana* Vol.31, No. 3, 2006, pp.1169-1174.

[6] L. Ciupală, A scaling out-of-kilter algorithm for minimum cost flow, *Control and Cybernetics* Vol.34, No.4, 2005, pp. 1169-1174.

[7] L. Ciupală and E. Ciurea, A highest-label preflow algorithm for the minimum flow problem, *Proceedings of the 11th WSEAS International Conference on Computers*, 2007, pp. 565-569.

[8] L. Ciupală and E. Ciurea, About preflow algorithms for the minimum flow problem, *WSEAS Transactions on Computer Research* vol. 3 nr.1, January 2008, pp. 35-41.

[9] L. Ciupală and E. Ciurea, An algorithm for the minimum flow problem, *The Sixth International Conference of Economic Informatics*, 2003, pp. 167-170.

[10] L. Ciupală and E. Ciurea, An approach of the minimum flow problem, *The Fifth International Symposium of Economic Informatics*, 2001, pp. 786-790.

[11] E. Ciurea and L. Ciupală, Sequential and parallel algorithms for minimum flows, *Journal of Applied Mathematics and Computing* Vol.15, No.1-2, 2004, pp. 53-78.

[12] E. Ciurea and L. Ciupală, Algorithms for minimum flows, *Computer Science Journal of Moldova* Vol.9, No.3(27), 2001, pp. 275-290.

[13] A. Deshpande, S. Patkar and H. Narayanan: Submodular Theory Based Approaches For Hypergraph Partitioning *WSEAS Transactions on Circuit and Systems*, Issue 6, Volume 4, 2005, pp. 647-655.

[14]V. Goldberg and R. E. Tarjan, A New Approach to the Maximum Flow Problem, *Journal of ACM* Vol.35, 1988, pp. 921-940.

[15] S. Fujishige, A maximum flow algorithm using MA ordering, *Operation Research Letters* 31, No. 3, 176-178, 2003.

[16] S. Fujishige, S. Isotani, New maximum flow algorithms by MA orderings and scaling, *Journal of the Operational Research Society of Japan* 46, No. 3, 243-250, 2003.

[17] S. Kumar, P. Gupta, An incremental algorithm for the maximum flow problem, *Journal of Mathematical Modelling and Algorithms* 2, No.1, 1-16, 2003.

[18] S. Patkar, H. Sharma and H. Narayanan: Efficient Network Flow based Ratio-cut Netlist Hypergraph Partitioning, *WSEAS Transactions on Circuits and Systems* vol. 3, no. 1, January 2004, pp. 47-53

[19] A. Schrijver, On the history of the transportation and maximum flow problems, *Mathematical Programming* 91, No.3, 437-445, 2002.

[20] K.D. Wayne, *A polynomial Combinatorial Algorithm for Generalized Minimum Cost Flow*, Mathematics of Operations Research., 445-459, 2002.