An Efficient Parallel Algorithm for Min-Cost Flow on Directed Series-Parallel Networks

Amit Jain N. Chandrasekharan*

Department of Computer Science
University of Central Florida
Orlando FL 32816

Abstract

We consider the problem of finding the minimum cost of a feasible flow in directed series-parallel networks. We allow real-valued lower and upper bounds for the flows on edges. While strongly polynomial-time algorithms are known for this problem on arbitrary networks, it is known to be "hard" for parallelization. We develop, for the first time, an efficient NC algorithm to solve the min-cost flow problem on directed series-parallel networks partially solving a problem posed by Booth and Tarjan [6, 5]. Our algorithm takes $O(\log^2 m)$ time using $O(m/\log m)$ processors on an EREW PRAM and it is optimal with respect to Booth and Tarjan's algorithm with running time $O(m \log m)$. The algorithm owes it's efficiency to the tree contraction technique and using simple data structures for flow list manipulations as opposed to finger search trees.

1 Introduction

Let $G = (V, E)$ be a directed network with two distinguished vertices s and t called the *source* and the *sink* respectively. For each $e = (u, v) \in E$, we define *source* $s(e) = u$ and *sink* $t(e) = v$. We always denote the number of vertices in G by n and the number of edges by m. With every edge $e \in E$ is associated three real-valued functions, a *lower bound l(e)*, a *capacity u(e)*, and a *cost c(e)*. A *flow* on a network is a real-valued function f on the edges satisfying the following constraints:

*Research supported in part by a National Science Foundation Grant No. NSF-CCR-9110159

capacity constraints

$$l(e) \leq f(e) \leq u(e) \text{ for all } e \in E$$

conservation constraints

$$\sum_{e \in E,\ t(e)=v} f(e) = \sum_{e \in E,\ s(e)=v} f(e) \text{ for all } v \in V - \{s,t\}$$

The value of the flow f is $\sum_{e \in E,\ t(e)=t} f(e)$, and the cost of f is $\sum_{e \in E} c(e)f(e)$. A flow is *maximum* if it has the maximum possible value. The *minimum-cost flow problem* is to find the minimum cost of a given feasible flow, among all flows with the same value. In particular, the *minimum-cost maximum-flow* problem is that of computing the minimum cost of the maximum flow from s to t. The *max-flow problem* is that of finding the maximum-flow from s to t.

1.1 Previous Work

The max-flow problem is a fundamental problem in combinatorial optimization and tremendous amount of work has been done for obtaining its algorithmic solution. Many polynomial-time algorithms are known for solving the minimum-cost flow problem (see for example [2, 10]). The best known bound for the running time is $O((m + n \log n)m \log n)$ [16]. Better bounds are known in special situations where the capacities or the costs are integers of moderate size. The min-cost flow problem is considered to be apparently "hard" for parallelization, because it has been shown in [11] that even the max-flow problem is P-complete. This implies that the existence of a polylogarithmic-time parallel algorithm using a polynomial number of processors (NC algorithm) is highly unlikely for this problem. Recently, it has been further proved [17] that approximating the value of the min-cost maximum-flow to within a constant factor is also P-complete. One way to cope with the apparent inherent sequentiality of these problems is to restrict them to special classes of networks. In [13], parallel algorithms for the maximum flow problem on planar networks have been presented which work either in $O(\log^3 n)$ time using $O(n^4)$ processors or in $O(\log^2 n)$ time using $O(n^6)$ processors. These algorithms have been later improved by [15], to $O(\log^3 n)$ time using $O(n^{1.5})$ processors. Contrasted with this, to the best of our knowledge, there are no known published NC algorithms for the max-flow min-cost problem on restricted classes of networks. Even in the sequential context, there has been at best only sporadic work on this problem. This is perhaps not so surprising, considering the polynomial-time sequential complexity of the problem. Bein, Brucker and Tamir [3] were the first to compute the minimum cost of the maximum flow on trees and series-parallel networks. For the special case of zero lower bounds, they observe that a simple greedy strategy works, resulting in a time bound of $O(nm + m \log m)$.

Two-Terminal Series-Parallel (TTSP) networks are a subclass of planar networks and can be defined recursively by the following rules:

1. $G = (\{s,t\}, \{(s,t)\})$, is a TTSP network. The vertices s and t are called the terminals. Let $G_1 = (V_1, E_1)$, with source s_1 and sink t_1, and $G_2 = (V_2, E_2)$ with source s_2 and sink t_2 be TTSP networks.

2. The network G_p formed from G_1 and G_2 by the *parallel composition operation* which identifies s_1 with s_2 and t_1 with t_2 is TTSP with source $s_1 = s_2$ and sink $t_1 = t_2$.

3. The network G_s formed from G_1 and G_2 by the *series composition operation* which identifies t_1 and s_1, is TTSP with source s_1 and sink t_2.

A network G is said to be *series-parallel* if there exist two vertices s and t such that G is a TTSP network with terminals s and t. In [6, 5], Booth and Tarjan present several improved algorithms for the general min-cost flow problem with lower bounds on the edge flows for series-parallel networks. Their first algorithm runs in $O(m \log m)$ time and $O(m)$ space if only the cost of a minimum cost flow is needed or $O(m \log \log m)$ space if the optimal flow distribution is needed. Their second algorithm for finding the actual flow distribution runs in $O(m \log m \log \log m)$ time but uses only $O(m)$ space. The problems of finding a linear-space algorithm with $O(m \log m)$ running time and obtaining a fast parallel algorithm have been left open in [5]. These algorithms work bottom-up on the *decomposition tree* representation of a series-parallel network. A *decomposition tree* of a series parallel network is a binary tree in which the leaves represent the edges of the network and each internal node represents either a series or a parallel composition of the networks represented by its subtrees. In Fig. 1, we show a series-parallel network and its decomposition tree.

The decomposition tree of a series-parallel network can be obtained in $O(m + n)$ time sequentially [18]. The best algorithm for recognizing a series-parallel network and obtaining a decomposition tree for it has been developed by Eppstein [9] which runs in $O(\log^2 n)$ time on the EREW PRAM with $C(m,n)$ processors; here $C(m,n)$ is the number of processors required to compute connected components of a graph in logarithmic time. The best known bound for $C(m,n)$ is $O(m\alpha(m,n)/\log n)$ [7].

1.2 Discussion and Merits of Our Work

In this paper, we present an efficient parallel algorithm for finding the cost of the minimum cost flow problem of a series-parallel network. We defer the problem of finding the actual flow distribution in parallel to future work. Given a decomposition tree, our algorithm runs in $O(\log^2 m)$ time using $O(m/\log m)$ processors on the EREW PRAM. The salient features of our algorithm are as follows:

- The spirit of our algorithm is from [6, 5] but the efficiency is due to a careful application of the tree-contraction technique [1]. The time-processor product of our algorithm matches the best known sequential time-complexity of Booth and Tarjan's algorithm and hence optimal.

- The algorithm computes a *flow list* at each internal node of the decomposition tree which represents the costs of minimum cost flows of all possible values for the subtree

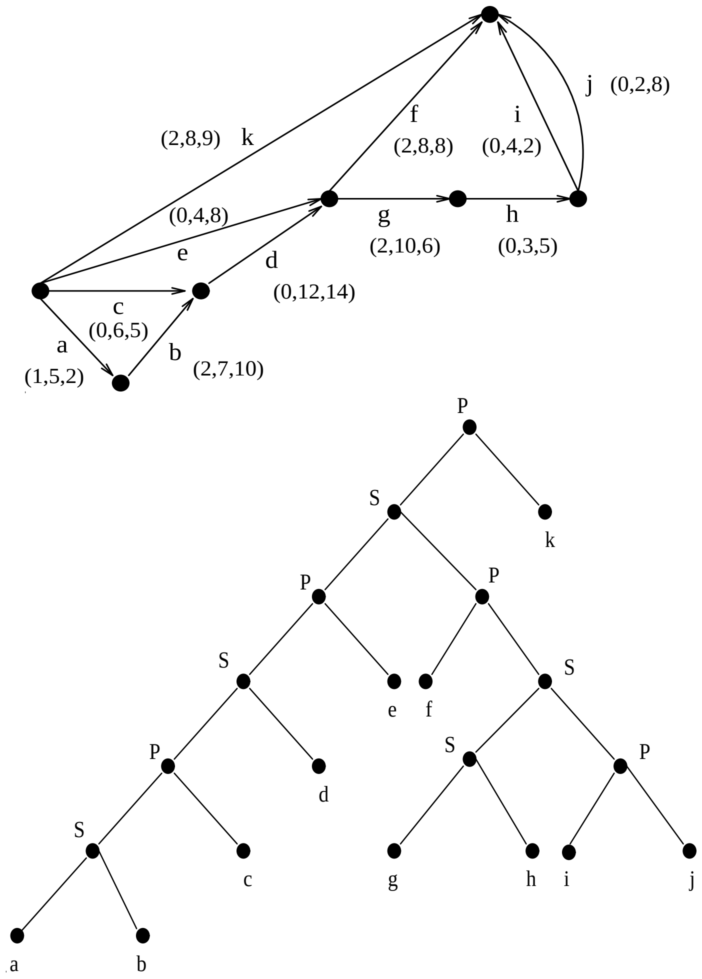

Figure 1: A series-parallel network and its decomposition tree. The edges are labeled by 3-tuples $(l(e), u(e), c(e))$.

rooted at that node. The size of the flow list is of $O(m)$ for a series-parallel network. The bottom-up procedure for obtaining the flow list for a node (from the flow lists of its subtrees) can take as much as $O(m)$ work resulting in an $O(m^2)$ work algorithm for the problem. To avoid this, Booth and Tarjan use an elaborate finger search tree representation of the flow lists with certain sophisticated operations and achieves an $O(m \log m)$ running time for her algorithm. Our parallel algorithm does not use the finger search tree and it is conceptually simpler. We believe that the sequential simulation of our parallel algorithm may perform better than Booth and Tarjan's in practice.

2 Min-Cost Flow on Series-Parallel Networks

2.1 Flow Feasibility

We can compute the maximum flow value and the minimum flow value on a series-parallel network G in $O(m)$ time sequentially using the decomposition tree. If the minimum flow value is greater than the maximum flow value there is no feasible flow. Let the maximum flow value be *maxval(G)* and the minimum flow value be *minval(G)*. Then the maximum and the minimum flow in G can be computed from the decomposition tree bottom-up, by the following equations:

base network Let $G = (\{s,t\},\{e\})$. Then $minval(G) = l(e)$, $maxval(G) = u(e)$.

Let G_1 and G_2 be series-parallel. Let their parallel composition be G_p and the series composition be G_s.

parallel composition

$$\begin{aligned} minval(G_p) &= minval(G_1) + minval(G_2) \\ maxval(G_p) &= maxval(G_1) + maxval(G_2) \end{aligned}$$

series composition

$$\begin{aligned} minval(G_s) &= max\{minval(G_1), minval(G_2)\} \\ maxval(G_s) &= min\{maxval(G_1), maxval(G_2)\} \\ \text{If } minval(G_s) &> maxval(G_s) \text{ then no flow is feasible.} \end{aligned}$$

The maximum and the minimum flow can be computed in parallel on an EREW PRAM using the tree contraction technique [1]. Briefly, the tree contraction approach reduces a binary tree by means of repeated *shunt* operations to a single node. The *shunt* operation involves pruning a leaf, eliminating its parent and setting its sibling's new parent to its previous grandparent. Due to space constraints, we skip the details of the algorithm and assume familiarity with it. We describe below, how to compute the maximum flow of G by providing the transformations associated with the shunt operation in the contexts of series

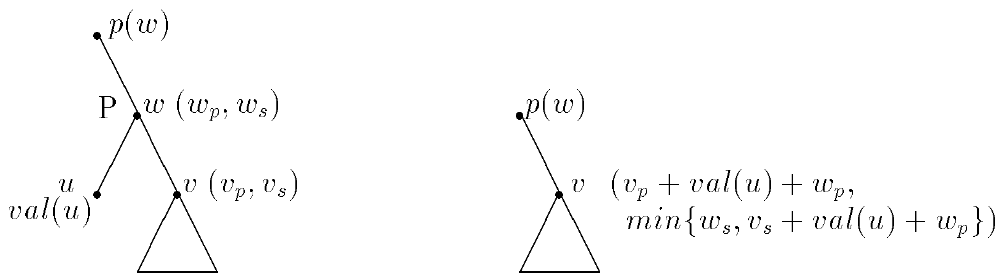

Figure 2: SHUNT on a parallel composition node.

and parallel composition operations.

With each parallel composition node in the decomposition tree, we associate the operation of *addition*, and with each series composition node we associate the operation of *minimum*. Each node x has a value $val(x)$, which is the maximum flow in the subgraph corresponding to the decomposition subtree rooted at x and a 2-tuple, say (a, b), associated with it. Then the actual value passed to the parent of x is $min\{val(x) + a, b\}$. Initially, each node is labeled with $(0, \infty)$. Each leaf node of the decomposition tree corresponding to an edge $e \in E$ is assigned $val(e) = u(e)$, the maximum flow value on that edge. The $val(y)$ for each internal node y is undefined at the start of the algorithm.

Fig. 2 shows a portion of a decomposition tree, illustrating the shunt operation on the leaf u composed in parallel with v at node w (shown as P). The value of the node v is $min\{val(v) + v_p, v_s\}$ where $val(v)$ is unknown. The value of the node u is $val(u)$. The value computed at node w is thus $val(u) + min\{val(v) + v_p, v_s\}$ which can be rewritten as $min\{val(u) + val(v) + v_p, val(u) + v_s\}$. In other words, the 2-tuple associated now with v is $(val(u) + v_p, val(u) + v_s)$. But we have the tuple (w_p, w_s) for w which should be combined with this, to form a new 2-tuple (z_p, z_s) for v such that $min\{val(v) + z_p, z_s\}$ is the same as (w_p, w_s) combined with $min\{val(u) + val(v) + v_p, val(u) + v_s\}$. A little manipulation yields (z_p, z_s) as equal to $(v_p + val(u) + w_p, min\{w_s, v_s + val(u) + w_p\})$. In the special case when $(w_p, w_s) = (0, \infty)$, we have $(z_p, z_s) = (val(u) + v_p, val(u) + v_s)$. If u is the right child, we can derive a similar expression. Also if the node w represents a series operation, then Fig. 3 shows the result after shunting u. For computing the minimum flow the two operations are *addition* and *maximum*. There we assign $(0, -\infty)$ to each internal node. We can carry out the analysis for the series operation just the same as above. Hence we have the following theorem using the results from [1].

Theorem 2.1 *Let $G = (V, E)$ be a series-parallel network with n vertices and m edges. If the decomposition tree for G is given, then the maximum flow, the minimum flow and feasibility of flow can be determined in $O(\log m)$ time using $O(m/\log m)$ processors on an EREW PRAM.*

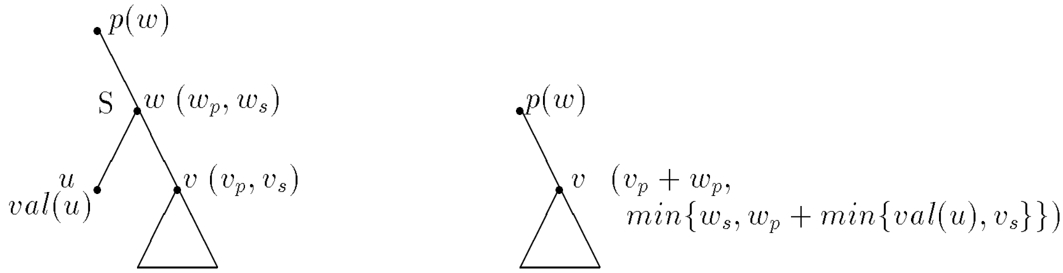

Figure 3: SHUNT on a series composition node.

2.2 Min-Cost Flow Value and Flow List Computations

The value of a maximum flow can be computed knowing only the maximum flow values of the constituent networks. However that is not true of the minimum cost of a maximum flow. To compute the minimum cost we need, in general, to know the costs of minimum cost flows of all possible values in the constituent networks. To represent this Booth and Tarjan introduced a *flow list*. A flow list is a list of pairs, sorted with respect to cost, representing the flow that can be pushed on the augmenting paths and the cost associated with it. Formally a flow list is a list of pairs $(l_0, c_0); (u_1, c_1), (u_2, c_2), \ldots, (u_k, c_k)$ with the following properties:

1. $u_i > 0$ for $1 \leq i \leq k$.
2. $c_i \leq c_{i+1}$ for $1 \leq i < k$.
3. l_0 is the value of the minimum flow, c_0 is the minimum cost of l_0 units of flow and $u_0 = l_0 + \sum_{i=1}^{k} u_i$ is the value of a maximum flow.
4. for any flow value $x, l_0 \leq x \leq u_0$, where $x = l_0 + \sum_{i=1}^{j-1} u_i + \alpha u_j$ with $0 < \alpha \leq 1$, the cost of a minimum-cost flow of value x is $c_0 + \sum_{i=1}^{j-1} c_i u_i + \alpha u_j c_j$.

The first pair is a *special* pair and the remaining pairs are *normal* pairs. The first component of a normal pair is the flow capacity and the second component is the cost. These pairs give the flow that can be pushed and the cost per unit flow for that flow, with each pair corresponding to an augmenting path in the network. Typically, the pairs with the same cost in a flow list are combined to reduce the size of the flow list. For general networks the flow list can be exponential in size. However for a series-parallel network the flow list has size $O(m)$ [6, 5].

A flow list for an arbitrary network can be constructed by first computing a minimum flow of minimum cost using any minimum-cost flow algorithm. This gives the special pair (l_0, c_0). Each subsequent pair is computed by finding a minimum-cost augmenting path in the residual network for the current flow. To construct a flow list for a series-parallel network G however, it is more efficient to build it by working on a decomposition tree of G bottom-up instead of using the minimum-cost augmenting path method. The following recursive formulation captures the essence of the algorithm.

base graph $G = (\{s,t\},\{e\})$. The flow list is $(l(e), c(e)l(e)); (u(e)-l(e), c(e))$ if $u(e) \geq l(e)$, else there is no feasible flow.

Let G_1 and G_2 be two series-parallel networks with flow lists $L_1 = (l'_0, c'_0); \bar{L}_1$ and $L_2 = (l''_0, c''_0); \bar{L}_2$ respectively.

parallel composition The flow list is $L_p = (l'_0 + l''_0, c'_0 + c''_0); \bar{L}_p$ where $\bar{L}_p$ is formed by merging lists $\bar{L}_1$ and $\bar{L}_2$, ordering the pairs in nondecreasing order of costs, and combining pairs of equal cost by adding their capacities.

series composition Let the flow list be $L_s = (l^s_0, c^s_0); \bar{L}_s$. First we have to compute the special pair. If $l'_0 = l''_0$ then $l^s_0 = l'_0$ and $c^s_0 = c'_0 + c''_0$. Otherwise we have to increase the minimum flow in the graph with smaller minimum flow to be equal to the minimum flow in the other graph. This may lead to deletion of some pairs in the flow list as they will be used to push the flow. Next we have to compute the remaining pairs of $\bar{L}_s$. Remove the first pair (u', c') from $\bar{L}_1$and the first pair (u'', c'') from $\bar{L}_2$. Add the pair $(min\{u', u''\}, c' + c'')$ to $\bar{L}_s$. If $u' > u''$ add the pair $(u' - u'', c')$ to the front of $\bar{L}_1$; if $u'' > u'$ add the pair $(u'' - u', c'')$ to the front of $\bar{L}_2$. Repeat this step until one of the two lists is empty.

The above description of the algorithm closely follows that of Booth [5]. In Fig. 4, a decomposition tree for the series-parallel network in Fig. 1 is shown along with all the intermediate flow lists. A straightforward implementation using linked lists to represent the flow lists, can compute a flow list for an m-edge series-parallel network using $O(m^2)$ amount of work. However, Booth and Tarjan obtained an $O(m \log m)$ time bound by using a clever representation of the flow lists using finger search trees. But, this is where the implementation of their algorithm becomes involved which in turn makes the analysis complex. Here we present a method which completely avoids the use of finger search trees. Although we describe our method in the parallel setting, it can be used profitably in the sequential setting as well. If we have the flow list for a graph then the minimum cost of a maximum flow (rather, of any flow) can be computed easily. From the flow list and a valid flow value k, we can compute the cost of the min-cost flow of value k as follows: Given flow list $(l_0, c_0); (u_1, c_1), (u_2, c_2), \ldots, (u_l, c_l)$ if $1 \leq j \leq l$ and $0 < \alpha \leq 1$ are such that $l_0 + \sum_{i=1}^{j-1} u_i + \alpha u_j = k$ then the cost of the min-cost flow of capacity k is $c_0 + \sum_{i=1}^{j-1} u_i c_i + \alpha u_j c_j$.

Now we introduce two operators: the parallel flow list operator $\odot_p$ and the series flow list operator $\odot_s$. These binary operators work on two flow lists and produce a new flow list corresponding to parallel or series composition operation. The identity element for the operator $\odot_p$ is the flow list $((0,0); \phi)$ where ϕ represents an empty list. The identity element for the operator $\odot_s$ is the flow list $((-\infty, 0); (\infty, 0))$. If there are no lower bounds on the flows on the edges, then the identity elements are ϕ and $(\infty, 0)$. In the following lemma we show some of the properties of these operators.

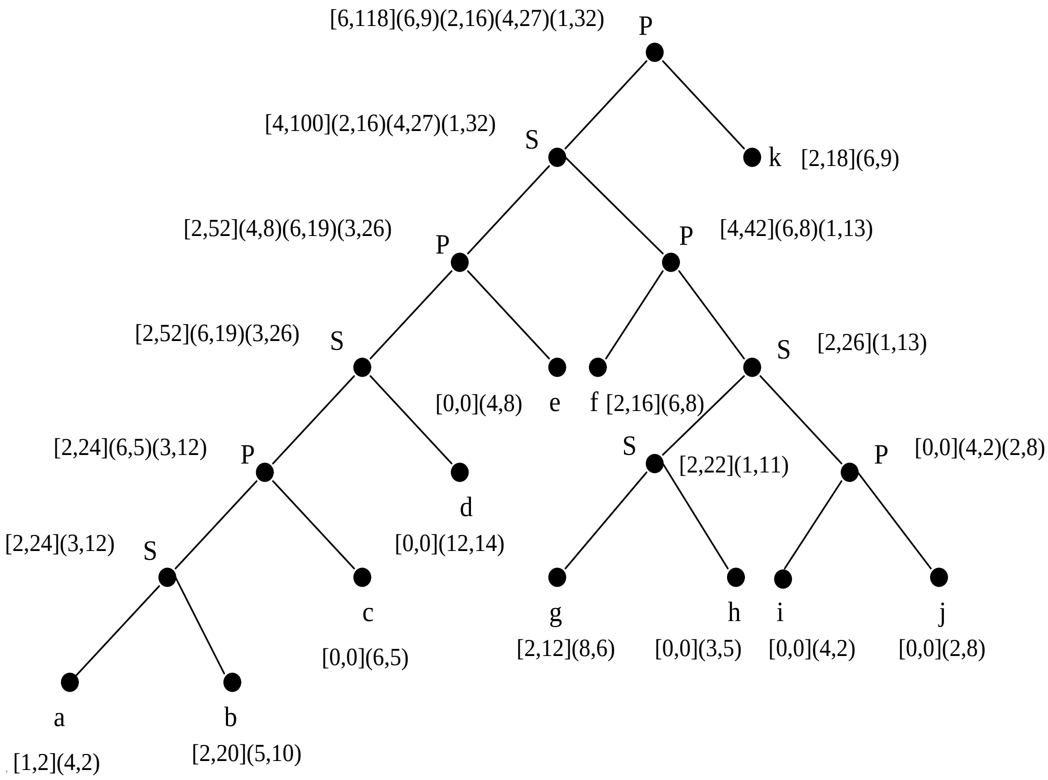

Figure 4: Flow lists for the series-parallel network in Fig. 1 (special pairs are shown in square brackets).

Lemma 2.2 *Series and parallel flow list operators have the following properties:*

1. $\odot_p$ *is commutative and associative.*
2. $\odot_s$ *is commutative and associative.*
3. $\odot_p$ *distributes over* $\odot_s$ *but not vice versa.*

Proof: Both parallel and series flow list operations are commutative by definition. Computation of the special pair for $\odot_p$ operator involves only addition which is associative. Combining the two flow lists uses straightforward merging which is also associative. Hence $\odot_p$ is associative. The series composition results in excess capacity of one of the constituent graphs being removed. Hence if we are combining three graphs in series the result depends only on which graph has the minimum capacity. Similarly computation of the special pair involves increasing the minimum flow to be the same as the largest minimum flow among the three graphs. Hence the $\odot_s$ operator is also associative.

For distributivity, let us consider $(A \odot_s (B \odot_p C))$. If $\odot_s$ distributes over $\odot_p$ then the resulting flow list should be the same for $((A \odot_s B) \odot_p (A \odot_s C))$. Suppose B had more capacity than C. Then we could push flow equal to the smaller of the capacity of A and C. But in the distributed expression we can push more flow. Hence the $\odot_s$ operator does not distribute over $\odot_p$ operator. The other case is similar.

Now, let us consider $(A \odot_p (B \odot_s C))$. If $\odot_p$ distributes over $\odot_s$ then the resulting flow list should be the same for $((A \odot_p B) \odot_s (A \odot_p C))$. The flow that can be pushed is the flow in A plus the minimum of the flow that can be pushed through B and C. If the distributed expression is considered, the flow is the the minimum of the flow through A and B and the flow through A and C which is the same as before. Hence the operator $\odot_p$ distributes over $\odot_s$. ♣

Our parallel algorithm for the min-cost flow problem follows a modified form of the tree contraction technique, where a shunt operation takes more than a constant amount of time because of the presence of flow lists of sizes $O(m)$. We associate a value with each node of the decomposition tree which is the flow list of the network rooted at that node. For each leaf node, corresponding to an edge e in the original network, the flow list is $((l(e), c(e)l(e)); (u(e) - l(e), c(e)))$, assuming flow is feasible. With each node we also associate a 2-tuple whose components are flow lists. Suppose for a node x, the flow list is L_x and the 2-tuple is (P_x, S_x). Then the flow list passed to the parent is $((L_x \odot_p P_x) \odot_s S_x)$. Initially, label each internal node of the decomposition tree by the 2-tuple of identity elements for the operators $\odot_p$, $\odot_s$, i.e., $(((0,0); \phi), ((-\infty, 0); (\infty, 0)))$.

Now consider the tree in Fig. 5, where the shunting is being done on node u. Then the value computed at node w is $(L_u \odot_p ((L_v \odot_p P_v) \odot_s S_v))$ which by Lemma 2.2 can be rewritten as $((L_v \odot_p (L_u \odot_p P_v)) \odot_s (L_u \odot_p S_v))$. Hence the new 2-tuple should be $(L_u \odot_p P_v, L_u \odot_p S_v)$. This 2-tuple has to be combined with the 2-tuple (P_w, S_w) which is discussed below in general terms. The case when u is the right son can be handled similarly. The case when the node w is a series composition node is also similar and the result is shown in Fig. 6.

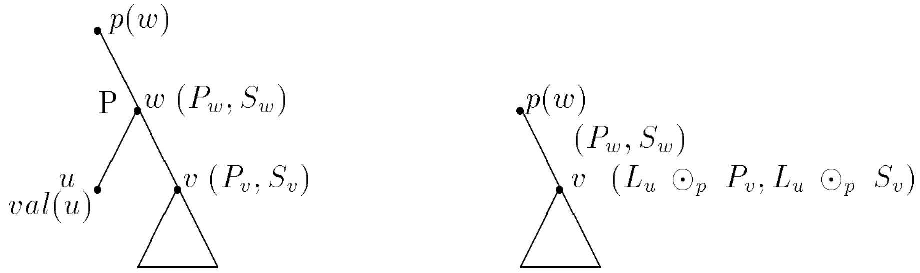

Figure 5: SHUNT on a parallel composition node.

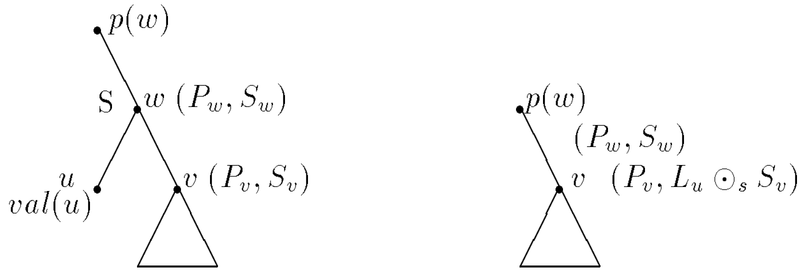

Figure 6: SHUNT on a series composition node.

We show here, how to combine two 2-tuples (P_x, S_x) and (P_y, S_y) where y is the parent of x such that L_x when applied to the resulting 2-tuple gives the same result as when applied to the two 2-tuples in succession. Using Lemma 2.2 we have:
$((((L_x \odot_p \ P_x) \odot_s \ S_x) \odot_p \ P_y) \odot_s \ S_y) \Rightarrow$
$(((((L_x \odot_p \ P_x) \odot_p \ P_y) \odot_s \ (S_x \odot_p \ P_y)) \odot_s \ S_y) \Rightarrow$
$((L_x \odot_p \ (P_x \odot_p \ P_y)) \odot_s \ ((S_x \odot_p \ P_y) \odot_s \ S_y)) \Rightarrow$
L_x applied to $(P_x \odot_p \ P_y, (S_x \odot_p \ P_y) \odot_s \ S_y)$

Hence we have the following theorem analogous to Theorem 2.1:

Theorem 2.3 *Given a series-parallel network with the decomposition tree the minimum cost maximum flow value and the flow list for the graph can be computed in* $O(\log m)$ *iterations using tree contraction.*

The time needed to perform an iteration of the algorithm is in the order of the time needed to compute the series or parallel composition of all the flow lists processed in any iteration. In the next subsection we will look at this problem in more detail.

2.3 Obtaining Flow Lists In Parallel

Suppose L_1 and L_2 are two flow lists of size m_1 and m_2 respectively. Let $m_1 \leq m_2$, and let $L_1 = (l'_0, c'_0); \bar{L}_1$ and $L_2 = (l''_0, c''_0); \bar{L}_2$. Assume that the flow lists are stored in arrays; later we will say how to maintain these flow lists. We prove the following lemmas.

Lemma 2.4 *Given two flow lists with m_1 and m_2 pairs, the composition of these two flow lists for the parallel composition operation can be done in $O(\log(m_1+m_2))$ time using $O((m_1+m_2)/(\log(m_1+m_2)))$ processors on the EREW PRAM.*

Proof: The special pair can be computed in $O(1)$ time by a single processor. The rest of the two flow lists can be merged in $O(\log(m_1+m_2))$ time using $O((m_1+m_2)/(\log(m_1+m_2)))$ processors on the EREW PRAM. Any optimal merging algorithm for the EREW PRAM can be used to accomplish this step [4, 8, 12]. After merging there may be up to two pairs with the same cost. These two pairs can be combined easily. We can then use the parallel prefix [14] algorithm to compact the array containing the flow list. ♣

Lemma 2.5 *Given two flow lists with m_1 and m_2 pairs, the series composition of these two flow lists can be done in $O(\log(m_1+m_2))$ time using $O((m_1+m_2)/(\log(m_1+m_2)))$ processors on the EREW PRAM.*

Proof: We provide the proof in two parts: (1) computing the special pair and (2) merging the remaining flow lists.

(1) Computing the special pair: If the minimum flow values are the same then computing the special pair is trivial and can be done in $O(1)$ time. Otherwise, without loss of generality, let $l'_0 > l''_0$ then we have to increase the flow in the second subgraph. This can be done in parallel as follows.

I.1 Compute in parallel, the prefix-sums on capacity u_i and cost $u_i c_i$ for the flow list L_2, i.e., $\sum_{i=1}^{1\leq k\leq m_2} u''_i$ and $\sum_{i=1}^{1\leq k\leq m_2} u''_i c''_i$.

I.2 Determine the index j of the pair and a fraction α such that $l'_0 = l''_0 + \sum_{i=1}^{j-1} u''_i + \alpha u''_j$, where $0 < \alpha \leq 1$.

I.3 Compute the cost as $c^s_0 = c'_0 + c''_0 + \sum_{i=1}^{j-1} u''_i c''_i + \alpha u''_j c''_j$.

I.4 Compress $\bar{L}_2$ to $((1-\alpha)u''_j, c''_j) \ldots (u''_k, c''_k)$ using parallel prefix.

All of the above can be done in $O(\log m_2)$ time using $O(m_2/\log m_2)$ processors on the EREW PRAM.

(2) Merging the remaining flow lists: This is a special type of merge operation where the merger of two pairs can result in at most two pairs with one added back to one of the flow lists being merged. The pair added back is the one with excess flow capacity. Sequentially the series operation can be accomplished in $O(m_1+m_2)$ time by modifying the standard merging algorithm. In the parallel setting, the key issue is how to partition the two flow lists into sublists such that these can be merged independently and are computationally balanced.

The idea is to partition the flow lists using the flow capacities and not just the cost. Note that the flow lists are ordered only by cost and not by capacity. We provide a high-level description of how to perform the series operation in parallel in the following steps:

II.1 Compute the prefix-sums on the flow capacities in the flow lists. (if not computed in Step I.1).

II.2 Partition the flow lists based on the cumulative flow capacity such that the corresponding sublists represent approximately equal flow capacity.

II.3 Make the flow capacity in corresponding sublists equal by splitting pairs in the adjacent sublists.

II.4 Merge the sublists independently using the sequential series operation.

Step 1 can be done in $O(\log(m_1+m_2))$ time in parallel. Suppose the total flow capacity is larger in one of the flow lists. Then we can discard the excess capacity in that list except one extra pair. The reason to keep one extra pair will become clear as we describe the rest of the steps. The prefix sums of the flow capacities for the two flow lists obviously form two increasing sequences. Let these two sequences be A (for L_1) and B (for L_2). To perform step 2 we will use these two sequences as follows.

II.2a Find the $i\lfloor\log(m_1+m_2)\rfloor$th ranked element in the union of the two sequences A and B, $i = 1, 2, \ldots, j (= \frac{(m_1+m_2)}{\lfloor\log(m_1+m_2)\rfloor})$. Let the output be two arrays $R_A[1..j]$ and $R_B[1..j]$, where $R_A[i] \neq 0$ implies that $A[R_A[i]]$ is the $(i\lfloor\log(m_1+m_2)\rfloor)$th element and $R_B[i] \neq 0$ implies that $B[R_B[i]]$ is the $(i\lfloor\log(m_1+m_2)\rfloor)$th element.

II.2b Let $R_A[0] = R_B[0] = 0$
for $i = 1, \ldots, j$ do
if $R_A[i] = 0$ then $R_A[i] = i * \lfloor\log(m_1+m_2)\rfloor - R_B[i]$
else $R_B[i] = i * \lfloor\log(m_1+m_2)\rfloor - R_A[i]$

Consider the flow sublists corresponding to the subsequences $A[R_A[i-1]+1]..A[R_A[i]]$ and $B[R_B[i-1]+1]..B[R_B[i]]$. The total flow capacity in one of the flow sublists may be smaller (the sequence which had $R_A[i]$ or $R_B[i]$ equal to zero at the end of Step II.2a). Without loss of generality, let this be the sublist of L_1 corresponding to A. Then we can split the $(R_A[i]+1)$th pair in the flow list L_1 into two pairs such that the flow capacity of sublist corresponding to A plus the new pair is exactly equal to that of the sublist corresponding to B. Note that we never need to split more than one pair else the value of $R_B[i]$ is not correct. Further, if we have $O((m_1+m_2)/\log(m_1+m_2))$ partitions of the flow lists L_1 and L_2, the sizes of the flow lists will increase by at most $O((m_1+m_2)/\log(m_1+m_2))$. Using parallel prefix, we can perform Step III and form new flow sublists which are exactly balanced in flow capacity. Then we can assign one processor to merge each flow sublist pair independently using the usual sequential algorithm.

The only other step we need to explain is Step II.2a. The ranking in Step II.2a can be done by merging the sequences A and B and then computing the positions. However, we can use a simpler approach by using the multiselection algorithm given in [8], by which we can avoid merging the two sequences. Given two ordered sets A and B of sizes m and n, where $m \leq n$, let us say we want to perform r selections where the ranks are $\{K_1, \ldots, K_r\}$. The algorithm given in [8] solves the problem in $O(\log m + \log r)$ time using r processors on the EREW PRAM and is used as a subroutine for developing an optimal merging algorithm on the EREW PRAM. The multiselection algorithm can be used to compute Step II.2a by setting $m = m_1$, $n = m_2$ and $r = (m_1 + m_2)/\log(m_1 + m_2)$. ♣

Remark 1. The sum of the sizes of all flow lists is initially at most m pairs (not counting the special pairs). Each parallel or series composition operation reduces or maintains the number of pairs. The various flow lists can be kept in an array of size $O(m)$ and in each iteration, we can keep track of the locations of the flow lists with appropriate bookkeeping. One way is to augment the decomposition tree such that each node also has pointers to the location and the size of its associated flow list. Hence each iteration of the tree contraction can be done in $O(\log m)$ time using $O(m)$ processors and $O(m)$ space. Thus the MCMF value can be computed in $O(\log^2 m)$ time using $O(m)$ processors on the EREW PRAM.

Remark 2. Using Brent's scheduling the number of processors can be reduced. The time taken for an iteration of the tree contraction is $max\ O(\log(m_{i_1} + m_{i_2}))$, $1 \leq i \leq k$, where k is the number of nodes in the decomposition tree being shunted in that iteration. We know that $\sum_{i=1}^{k} m_{i_1} + m_{i_2} = O(m)$. A simple strategy is to allocate $\lceil m_i/\log m \rceil$ processors to a flow list of size m_i in each iteration of the tree-contraction, instead of $\lceil m_i/\log m_i \rceil$ processors. Hence the total number of processors is bounded by $O(\sum_{i=1}^{k} \frac{m_{i_1}+m_{i_2}}{\log m})$ and the time by $O(\log m)$ in each iteration. Hence we can reduce the number of processors to $O(m/\log m)$ without increasing the asymptotic running time, giving rise to our main theorem:

Theorem 2.6 *Given a series-parallel network with n vertices and m edges and its associated decomposition tree, the minimum cost of any flow can be computed in $O(\log^2 m)$ time using $O(m/\log m)$ processors on the EREW PRAM.*

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. *SIAM Review*, 33(2):175–219, June 1991.

[3] W. W. Bein, P. Brucker, and A. Tamir. Minimum cost flow algorithms for series-parallel networks. *Discrete Applied Mathematics*, 10:117–124, 1985.

[4] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. *SIAM Journal on Computing*, 18(2):216–228, April 1989.

[5] H. Booth. Fast algorithms on graphs and trees. Technical Report 90-76, Center for DIMACS, Rutgers University, December 1990. (also Ph.D. Thesis, Princeton University, 1990).

[6] H. Booth and R. E. Tarjan. Finding the minimum-cost maximum flow in a series parallel network. *Journal of Algorithms*, 1993. to appear.

[7] R. Cole and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, May 1991.

[8] N. Deo, A. Jain, and M. Medidi. Parallel multiselection and optimal parallel merging. In *Proceedings of Thirtieth Annual Allerton Conference on Communication, Control, and Computing*, pages 215–224, 1992.

[9] D. Eppstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98:41–55, 1992.

[10] A. V. Goldberg, Éva. Tardos, and R. E. Tarjan. Network flow algorithms. Technical Report CS-TR-216-89, Princeton University, March 1989.

[11] L. Goldschlager, L. Shaw, and J. Staples. The maximum flow problem is log space complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[12] T. Hagerup and C. Rub. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, December 1989.

[13] D. Johnson. Parallel algorithms for minimum cuts and maximum flows in planar networks. *Journal of the ACM*, 34:950–967, 1987.

[14] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[15] G. Miller and J. Naor. Flow in planar graphs with multiple sources and sinks. In *IEEE Foundations of Computer Science*, volume 30, pages 112–117, 1989.

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

[17] C. Stein and J. Wein. Approximating the minimum-cost maximum-flow is P-complete. *Information Processing Letters*, 42:315–319, 1992.

[18] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, May 1982.