Parallel Computing for Network Optimization: A Cluster-Based Approach for the Dual Transshipment Problem

Raghu P. Chalasani

Cadence Design Systems, Inc.
555 River Oaks Parkway, MS 2A2
San Jose, CA 95134

K. Thulasiraman

University of Oklohoma
200 Felgar Street, Room 116
Norman, OK 73019-0631

Abstract

The traditional simplex method for solving the transshipment problem or its dual [2] does not offer much scope for parallelization because it moves from one basic solution to another. To address this problem, we recently presented a new method [9] called the Modified Network Dual Simplex method. This method incorporates two novel features: a technique to convert a non-basic dual feasible solution to a basic dual feasible solution, and strategies for performing pivots concurrently. In a more recent paper [1], the suitability of this approach in the Integrated VLSI layout compaction and wire balancing problem was discussed. In this paper, we describe another novel method for solving the dual transshipment problem. It combines the concurrent pivoting strategies of [9] with a massively parallel method for converting a non-basic feasible solution to a basic feasible solution (without reducing the objective value). The method employs extensively the notion of cluster firing. Results of testing of this method on large scale graphs are also presented.

1 Introduction

Network optimization refers to the class of optimization problems defined on graphs. These problems include the problem of constructing shortest paths, finding a maximum flow, constructing a min-cost spanning tree, finding matchings in networks etc. These problems occur in a variety of applications. While they are themselves significant in their usefulness, they also occur as subproblems in several applications.

Kruskal's and Prim's algorithms for the min-cost tree problem, Dijkstra's and Bellman-Ford-Moore's algorithms for the shortest path problems, Ford-Fulkerson's algorithm and its several variants for the maximum flow problem are among the most fundamental algorithms in network optimization theory [12]. They have also served as the basis for designing corresponding distributed/parallel algorithms.

The **transshipment problem** which can be formulated as a linear programming problem is a general class of network optimization problem [2]. Several network optimization problems such as those mentioned above are special cases of the transshipment problem. Many problems which occur in engineering and industrial applications can be formulated either as a transshipment problem or as its dual. For instance, VLSI layout compaction and wire balancing can be formulated as a dual transshipment problem [7], [13]. There are two basic approaches to the transshipment problem - the **network simplex method** and the **network primal dual method** [2]. Goldberg [5] presented a variant of the primal-dual method called the ϵ-relaxation method. References to other ϵ-relaxation methods may also be found in [5]. These relaxation methods have been designed primarily to achieve good complexity results for the transshipment problem. In view of the practical importance of the transshipment problem, there has been considerable interest in designing distributed/parallel algorithms for this problem. Peters [8] has presented a parallel implementation of the network simplex method. Goldberg's algorithms in [5] are also suitable for parallel implementation. We have discussed in [10] a parallel implementation of the network primal-dual method. Recently, we have presented in [9] the Modified Network Dual Simplex method and its parallel implementation. An interesting property of this algorithm useful in the context of the integrated layout compaction and wire balancing problem is presented in [1].

In this paper, we describe development of a novel parallel algorithm for the dual transshipment problem. This algorithm extensively employs the notion of cluster firing [3], [11] and the strategies developed in [9] for efficiently performing pivot operations concurrently.

The paper is organized as follows. In Section II, certain basic definitions are presented. The dual transshipment

1063-6374/95 $04.00

problem and the network dual simplex method to solve this problem are reviewed in Section III. In Section IV, a new characterisation of the optimum solutions of the dual transshipment problem is presented. In Section V, a new approach called Cluster-Based Dual Simplex method to solve the dual transshipment problem is outlined. Several issues relating to the implementation of this new approach are discussed in Section VI. Experimental results on the implementation of this algorithm are presented and discussed in Section VII.

2 Preliminaries

A **directed graph** G with node set V and edge set E will be denoted by G = (V, E). The nodes will be denoted by the integers 1, 2, ..., n. Thus V = {1, 2, ..., n}. An edge e connecting nodes i and j and directed from i to j will be denoted by $e = (i, j)$. We assume that G is connected. Each node i will be associated with a real number w_i, called the **weight** of i. Each edge $e = (i, j)$ will be associated with a real number m_{ij} called the **token** of e. W will denote the row vector of node weights and M_0 will denote the column vector of edge tokens. A node j is an **outnode** at node i, if (i, j) is an edge in G. Similarly, a node k is an **innode** at node i, if (k, i) is an edge in G. A subset of nodes will be called a **cluster**. Given $S \subset V$, $\overline{S} = V - S$ will refer to the **complement of S** in V. $(S, \overline{S})$ will refer to the set of edges connecting the nodes in S with those in $\overline{S}$. Thus if $(i, j) \in (S, \overline{S})$, then either $i \in S$ and $j \in \overline{S}$ or $i \in \overline{S}$ and $j \in S$.

Given a spanning tree T of G. Consider an edge $e = (i, j)$ of T. Removing e from T will result in two connected subgraphs T_1 and T_2 with node sets V_1 and V_2. Note that $V_1 = V - V_2$. Then (V_1, V_2) will be referred to as the **fundamental cutset** with respect to the edge e of T, and V_1 and V_2 will be referred to as **fundamental clusters**.

The notion of firing [3], [11] will be used extensively in the development of our algorithm in this paper. **Positive firing x times of a node** i, is the operation of adding x to the token of every outgoing edge (i, j) and subtracting x from the token of every incoming edge (j, i). **Negative firing x times of a node** i, is the operation of adding x to the token of every incoming edge (j, i) and subtracting x from the token of every outgoing edge (i, j). Firing a cluster x times results in firing every node in the cluster x times. The **firing number** of a node is the number of times this node has been fired. After a positive (negative) firing of a node, the node firing number is increased (decreased) by the appropriate number.

A firing, positive or negative, should not cause any edge token to be negative. Positive firing is the most commonly used notion in the theory of marked graphs [3], [11]. Hence, unless otherwise stated, positive firing will be referred to simply as **firing**.

In our model of parallel computation, we assign to each process exactly one node of the graph under consideration. Communication among processes is through shared variables. We permit concurrent reads. No more than one process can write into a shared variable. Lock variables are used to achieve this. Number of processes is not restricted to be equal to the number of processors available on a parallel machine. In our algorithm, during a **pulse** all processes execute the same set of instructions. Some of the processes may be idle. The actions taken by the processes during a pulse may vary from one process to another and will depend on the values of certain variables at the beginning of the pulse. A **phase** in the algorithm will usually consist of several pulses. Our algorithms have been implemented on the Shared memory BBN Butterfly machine.

3 The Dual Transshipment Problem (DTP) and the Network Dual Simplex Method

Consider a directed graph G = (V, E) with node weight vector W and edge token vector M_0. Let A denote the incidence matrix of G [12] and A^t be its transpose. The dual transshipment problem (DTP) is a linear program defined as follows:

Maximize: $W^t Y$
subject to

$$A^t Y \geq -M_0 \quad (1)$$

$$Y \geq 0 \quad (2)$$

In the above, Y is a column vector of node variables, called **firing numbers**. Thus y_i will denote the firing number of node i. For each edge (i, j) inequality (1) contains a contraint of the form

$$y_i - y_j \geq -m_{ij}.$$

$y_i - y_j + m_{ij}$ is thus the value of the **slack variable** associated with the edge (i,j). If we fire each node i, y_i times, then the new token on edge (i, j) will be $y_i - y_j + m_{ij}$. This interpretation suggests the name **residual token** for the expression $y_i - y_j + m_{ij}$. Thus the dual transshipment problem is to obtain a vector Y such that

1. $W^t Y$ is maximum, and
2. the residual token on every edge is non-negative if the nodes are fired as specified by the firing numbers y_i's.

A vector Y is called a **feasible solution** if Y satisfies constraint (1). A vector Y is called a **basic feasible solution** if G has a spanning tree T such that the residual tokens of all edges of T become zero when the nodes are fired as specified by the node firing numbers y_i. The tree T is then called a **basic feasible tree** corresponding to the

basic feasible solution Y. Such a tree T will also be called a **0-token spanning tree.**

Since during firing no residual token should become negative, it follows that a node i can be fired at most y_i times where y_i is the smallest residual token on any incoming edge (j, i). Note that if residual tokens permit independent firings of nodes i and j, then these firings can be done concurrently without making any resulting residual token negative. It is this property that we take advantage of in developing our parallel algorithm.

Let Y be a basic feasible solution and T be the corresponding basic feasible tree. Consider an edge (i, j) and let $(S, \overline{S})$ be the corresponding fundamental cutset (see definitions in the previous section) with $i \in S$ and $j \in \overline{S}$. Recall that S and $\overline{S}$ are called **fundamental clusters** with respect to edge (i, j) of tree T. A **pivot operation** is permissible if $W(S) \geq 0$ where $W(S) = \sum_{i \in S} W(i)$. A pivot operation consists of firing the cluster S the maximum possible number of times and constructing the new basic feasible tree. Note that firing S results in a new Y vector and new residual tokens. It can be shown that the new Y is also a basic feasible solution.

The **network dual simplex** (NDS) method is essentially the simplex method of linear programming [2] applied to solve the DTP. Thus the method consists of the following steps.

1. Construct an initial basic feasible solution Y_0, if it exists. This is achieved by constructing an auxiliary network and applying the simplex method on this network. This step detects infeasibility of the DTP, whenever that is the case.
2. Perform a pivot operation.
3. Check the new basic feasible solution for optimality [2]. If it is optimal, then the algorithm terminates. Otherwise, repeat step (2) starting from the current basic feasible solution.

Unboundedness of the DTP is detected if we encounter a fundamental cluster S such that $W(S) > 0$ and every edge in $(S, \overline{S})$ is directed from a node in S to a node in $\overline{S}$. Note that in such a case the cluster S can be fired an unbounded number of times leading to an unbounded value for the objective $W^t Y$. To avoid cycling, Bland's anti-cycling rule [2] can be used in step (2) while selecting a pivot operation. For all details relating to the simplex method, [2] may be consulted.

4 A Characterisation of Optimum Solutions for the DTP

Consider a directed graph G defining a DTP. Let V_n be the set of negative weight nodes in G. Then, given a feasible solution to the DTP, *cluster* C_i, for each negative node $i \in V_n$, is defined as the set of nodes reachable from i through 0-token directed paths. The negative node i is called the *source* of the cluster C_i. The *weight of the cluster* C_i is the sum of the weights of all the nodes in that cluster.

Clusters C_i and C_j are said to be *mutually exclusive* if $C_i \cap C_j = \emptyset$. Three or more clusters are *mutually exclusive* if every pair of them are also mutually exclusive. Clusters C_i and C_j are said to be *linked* if $C_i \cap C_j \neq \emptyset$.

A collection S_k of clusters is said to be maximally linked,

1. if a cluster $C_i \in S_k$, then all clusters that are linked to C_i are also in S_k, and
2. if a cluster $C_j \notin S_k$, then $C_i \cap C_j = \emptyset$, for all $C_i \in S_k$.

We now present an alternate characterisation of optimality in terms of the clusters defined above.

Theorem 1

A solution to the DTP is optimum iff the corresponding clusters satisfy the following properties:

P1. For each negative node i, the weight of the cluster C_i is non-negative.

P2. The weight of the union of two or more clusters is non-negative.

Proof

Necessity

Suppose a solution Y is optimum and the corresponding clusters do not satisfy (P1.) or (P2.). Then there would exist a group S of nodes with $W(S) < 0$. Since the edges going out of each cluster have non-zero residual tokens, it follows that S can be fired negatively a non-zero number of times, thus improving objective $W^t Y$. This contradicts the optimality of Y. Thus the clusters corresponding to Y satisfy (P1.) and (P2.).

Sufficiency

Suppose that the clusters corresponding to a solution Y satisfy (P1.) and (P2.). We show that for every group S of nodes with $W(S) < 0$, there is a zero token edge going out of S. This could then mean that no further negative firing of nodes is possible and hence the objective $W^t Y$ cannot be improved any more (establishing the optimality of Y).

Consider a group S of nodes with $W(S) < 0$. Let $v_1, v_2, \ldots, v_k$ be the negative nodes in S and $C_1, C_2, \ldots, C_k$ be the

corresponding clusters. Suppose all these clusters lie entirely in S. By (P2.), the union of these clusters have non-negative weight. Since the negative nodes v_1, v_2, ..., v_k are all in this union, the weight of S will be non-negative contradicting that W(S) < 0.

If, on the other hand, C_i - S ≠ ∅, for some C_i, $1 \le i \le k$, then there exists an edge (v_a, v_b) with zero residual token and with $v_a \in S$ and $v_b \notin S$. Thus, for every group S of nodes with W(S) < 0, there is a zero-token edge going out of S. ❑

5 CBDS: A Cluster-Based Dual Simplex for the DTP

The optimality criteria proved in Theorem 1 has the following algorithmic implication.

Suppose there are k negative nodes and hence k clusters which satisfy (P1.). Then to test for property (P2.), we need to generate all the 2^k - 1 combinations of clusters and check if any one of these combinations is negative. The solution is optimum if all of them are non-negative. An approach based on this optimality criteria will be attractive only for values of $k \le 3$. So, our approach to be presented next will not employ the test for property (P2.) though it will be based on clusters which satisfy property (P1.). For this reason, we shall refer to this approach as the Cluster Based Dual Simplex (CBDS) method.

The main steps of the CBDS method for the DTP are:

1. Feasibility Testing.
2. Cluster Forming.
3. Cluster Optimization.
4. Cluster Union.
5. Firing Zero Combinations.
6. Concurrent Pivots.

We first test if the DTP is feasible (Step 1). The algorithm would terminate if the DTP is not feasible. Otherwise, we form clusters (Step 2). If any of these clusters has negative weight, then we fire these clusters in an appropriate manner until all the clusters have non-negative weight (Step 3).

If, at the end of Step 3, the subgraph of zero residual tokens is not connected, then each connected component will correspond to a set of maximally linked clusters. In Step 4, we combine these clusters, and treating each such combination as a cluster, we return to Step 3.While optimizing these combinations, it may so happen that some of these combinations may be reduced in size. When this happens, the clusters inside these combinations may not be connected with a zero-token edge. So, to avoid this case, if any combination is reduced in size, the algorithm will go back to Step 2 to form new clusters.

If, at the end of Step 4, the weight of the sum of the nodes in every connected component of zero-token edges is zero, then we fire these components in an appropriate manner to create a connected spanning subgraph of zero-token edges (Step 5).

At this point, a 0-token spanning tree will be available. Using this tree, we perform in Step 6 concurrent pivots as described in [9].

Step 5 will not be required if, at the end of Step 4, the subgraph of zero-token edges is connected and spans all the nodes in the graph.

Steps 2-6 would be repeated if optimality is not detected at Step 6.

These steps will be discussed in detail in the following sections.

6 Implementation of CBDS

6.1 Feasibility Testing

We test the feasibility of the DTP by applying algorithm FEASIBLE discussed in [4]. If the problem is not feasible, then the algorithm stops here; otherwise it will proceed to the next step. Note that at the end of this step, if the DTP is feasible, then the residual edge tokens will all be non-negative.

6.2 Cluster Forming

In this step, a cluster C_i is formed for each negative node $i \in V_n$. It involves finding for each negative node i, the set of all nodes reachable from i by directed paths with zero residual tokens. A cluster can also be viewed as a 0-token subtree rooted at each negative node. A cluster can be overlapping with other clusters either completely or partially.

Clusters are only a 'soft' grouping of nodes i.e., the nodes in a cluster are not contracted into one single node as in our previous algorithm. They still keep their individual identities and are aware of all the clusters in which they are present.

There are two phases in this step. The first one, called Cluster Initialization, will be used to form a initial cluster at each negative node consisting of only itself. The second phase, called Cluster Expansion, will be used to expand each cluster as much as possible. The following data structures are used to explain the details of these phases.

token : a 2-dimensional array of integers. *token [i] [j]* represents the value of the token of the edge (i, j), if there is such an edge in the original graph

G; otherwise it contains a special value INFINITY outside the range of edge tokens.

firingNo : an array of integers. At any time, *firingNo [i]* represents the current firing number of node *i*.

nodeWeight : an array of integers. *nodeWeight [i]* represents the weight of the node *i*.

clusterWeight : an array of integers. Each *clusterWeight [i]* is initialized to *nodeWeight [i]* which is the weight of node *i*. At termination, *clusterWeight [i]* represents the weight of the cluster, if node *i* is the source of the cluster..

sources : an array of sets. The variable *sources [i]* contains the set of the sources of all the clusters in which node *i* is present.

newSources : an array of sets. The variable *newSources [i]* contains the set of the new sources inviting node *i* to join their clusters.

6.2.1 Cluster Initialization

There is only one pulse in this phase. Each node keeps track of the clusters in which it is present by using a set variable called *sources*. Each *sources [i]* will contain all the sources in whose clusters node *i* is present. To initialize the process, each negative node *i* will add its node number *i* to its *sources [i]* and its node weight to *clusterWeight [i]*.

6.2.2 Cluster Expansion

In the first pulse of this phase, each node will calculate its zero outnodes and propagate its *sources* information to any new zero outnodes. In the subsequent pulse, each node will examine its *newSources* variable and if any new sources have been added to it since the last iteration, then it will propagate this new sources information to each of its zero outnodes. It will also update the cluster weights of new sources by adding its node weight to the weight of each of their clusters. This pulse is repeated until there is no new propagation at any node in one pulse.

The above implementation can be done in an asynchronous manner. Also, each node doesn't have to wait or synchronize for any particular data; it propagates as and when it 'sees' it.

6.3 Cluster Optimization

There are two phases in this step. The first phase is to fire those clusters whose weights are negative. The second phase is to expand the clusters that have been fired in the first phase. These two phases will be repeated until all the clusters in the graph have become non-negative weighted. We will explain each of these phases in detail in the following subsections.

6.3.1 Cluster Firing

There are four pulses in this phase. The first pulse is used to check if there is any negative weighted cluster in the graph. If not, then this phase is terminated and the algorithm proceeds to the Cluster Union step.

The firing number for each cluster is calculated in the second pulse. To find the firing number of a cluster C_j, each node in that cluster will examine its outgoing edges one by one. It will examine only those outgoing edges that are also going out of the cluster C_j and pick the minimum residual edge token. Note that the outgoing edges at a node *i* that are going out of the cluster C_j may not be the same as the ones that are going out of the cluster C_k if node *i* is present in both the cluster C_j and the cluster C_k. So, a node may have to process its outgoing edges more than once if it is present in more than one cluster.

The minimum residual edge token computed by a node in Cluster C_j will be the maximum number of times this cluster could be fired. After computing this firing number, it will compare this value with that of the source and make the smaller of these two as the new firing number of the source. This process is repeated by each node in the cluster. Thus, the most constraining firing number of all the nodes in the cluster C_j will be written as the cluster firing number at the source.

In the third pulse, each negative node will examine its cluster weight. If the cluster weight is non-negative, then that cluster is not permitted to fire unless if it is part of another negative cluster. Therefore, each negative node will make its cluster firing number zero if the weight of its cluster is non-negative and it is not part of any negative cluster. If its cluster weight is non-negative and it is part of negative cluster(s), then it will initialize its cluster firing number to the greatest of the firing numbers of the negative clusters which it is part of.

In the fourth pulse, each node *i* will get the cluster firing number from each of its sources. The maximum of these numbers is its firing number and it will fire by that amount. Thus, if a node *i* is present in more than one cluster, then it will fire along with the cluster whose cluster firing number is greater than or equal to the cluster firing

numbers of the other clusters in which it is present. After this firing, node i will not be part of those clusters whose firing numbers are less than its firing number. The weights of these clusters will be adjusted accordingly. This also means a cluster which has non-negative weight in one iteration may become negative weighted and need to participate in the subsequent iterations of these pulses. One can easily verify that this scheme will not make the residual edge tokens of any node negative.

Suppose a cluster is negative weighted and there is no edge going out of this cluster, then this cluster can fire ∞ times which implies that the problem is unbounded. This can easily be detected in this phase and the algorithm will exit with appropriate error messages.

6.3.2 Cluster Expansion

Every cluster that has been fired will create at least one 0-token outgoing edge from the cluster. In the second phase, clusters will expand along with these newly created 0-token edges.This phase is the same as the one explained in Section 6.2.2.

The cluster optimization step is summarized below.

while (there is at least one cluster with negative weight) **do**

a) Fire Negative Weight Clusters

b) Expand Clusters

end while

6.4 Cluster Union

When the algorithm enters this step, there is a cluster rooted at each negative node and the weight of every such cluster is non-negative. Thus, the clusters satisfy property (P1.) in Theorem 1.

In the Cluster Union Step, clusters which are overlapping with each other are combined into one. If there are more than one combinations at the end of this step, then the algorithm goes back to the Cluster Optimization step at the end of which all the combinations are non-negative weighted.

There are four pulses in this step like in the previous step. In the first pulse, all positive nodes with no zero outnodes will examine their source variables. If they have more than one source, then they pick the least numbered source as the root and inform every one in the source variable about this root. They use a variable *combination* to propagate this root information. *combination* is an array of sets. The variable *combination [i]* contains the set of the sources that should be combined with the source i. Also, each source (negative node) i will re-initialize its *clusterWeight [i]* to zero.

In the second pulse, each source will check its *combination* variable and propagate the root information to other sources in that combination. This pulse will be repeated until all nodes complete the propagation. At the end of this pulse, all sources will be aware of the root of the combination.

In the third pulse, each node will get the root information from their sources and add its weight to the *clusterWeight* of the root. Each node i will also re-initialize its *sources [i]* to root.

The fourth pulse will be used to check if there is any negative combination formed in this step. If so, then the algorithm will go back to the Cluster Optimization step to fire and expand those negative combinations. Otherwise, it will proceed to the next step: Firing Zero Combinations. In this pulse, each root will also increase by 1 the *no_of_clusters* variable so that the algorithm will know if there are more than one combination when it goes to the Firing Zero Combinations phase.

6.5 Firing Zero Combinations

If the number of combinations formed at the end of the Cluster Union step is more than one and if the weight of each such combination is non-negative, then each of these combinations has zero weight. This is because we are assuming that the sum of the weights of all the nodes in the graph is zero. At this stage, the algorithm has to use the Concurrent Pivots step to check the optimality of the solution, but there may not be any 0-token spanning tree available. So, in the Firing Zero Combinations step, each zero combination is fired appropriately to merge with other zero combination(s), until all the nodes are coalesced into one combination.

There are, again, four pulses in this step. In the first pulse, each zero combination will calculate its firing number as explained in the Cluster Optimization step. It will also find out the root of the other combinations they are going to merge with, if it fires by its firing number.

In the second pulse, each combination will check if it is allowed to fire. If cluster i is allowed to fire and if it is going to merge with cluster j, then cluster j is not allowed to fire. This is decided on a first-come first-serve basis.

In the third pulse, each cluster i that is allowed to fire will fire by the amount it calculated in the first pulse.

In the fourth pulse, each cluster that is fired in the previous pulse will merge with the other cluster j. Again, the smaller of (i, j) will become the root of that combination. Since these are non-overlapping clusters, the weight of the combination will be sum of the weights of the clusters.

6.6 Concurrent Pivots

At the end of Step 5 (Firing Zero Combinations), a 0-token spanning tree is available and we have a basic feasible solution. So, the algorithm builds a 0-token spanning tree and tests it for optimality using the Concurrent Pivots step (Step 6). This step described in [9] is explained below for the sake of completeness. If the solution is not optimum, then the algorithm goes back to the Cluster Forming step.

Given a basic feasible solution Y and the corresponding basic feasible tree, concurrent pivoting essentially involves traversing the tree bottom up (starting from the leafs), identifying the fundamental clusters and firing them in an appropriate manner.

We assume that the following data structures are available at each node.

For each node i, *father [i]* will denote the unique father of i obtained by a depth-first-search of T. At the end of a cluster finding pulse, *origin [i]* will denote the root of the fundamental cluster which contains node i (Note that this fundamental cluster will also define a unique subtree of T rooted at *origin [i]*.) and *clusterWeight [i]* will denote the weight of this cluster. Initially, *clusterWeight [i]* = *nodeWeight [i]*. Similarly, *clusterFringNumber [i]*, at the end of the cluster finding pulse, will denote the minimum residual token of all edges (j, i) where node j is not a member of the cluster in which node i is present. This variable is initialized to INFINITY. *treeNodes [i]* will denote the set of nodes adjacent to i in T. Initially, *num_tree_edges [i]* will denote the number of nodes that are adjacent to i in T. As the bottom-up traversal of T proceeds, the edges will be contracted and the tree will shrink dynamically.

Concurrent pivoting involves two phases: cluster firing and determining new clusters.

Cluster Firing: In this phase, there are two subpulses: the positive cluster firing pulse and the negative cluster firing pulse.

Positive Cluster Firing: During the positive firing pulse, only nodes i with *num_tree_tree [i]* ≤ 1 and *clusterWeight [i]* > 0 will be active. Each such node i will process the edges in *tree_nodes [i]* one by one and discard those edges (j, i) with *origin [j]* = *origin [i]*. From the remaining edges, it will calculate the minimum residual edge token, say, f_i. After calculating f_i, node i will lock the *clusterFiringNumber* variable of *origin [i]* and will write

$$clusterFiringNumber\ [origin\ [i]] = f_i, \text{ if } f_i < clusterFiringNumber\ [origin\ [i]].$$

At the end of this pulse, *clusterFiringNumber [origin [i]]* will give the number of times each node in the cluster with *origin [i]* as root can be fired positively. Each node i will find its firing number from *clusterFiringNumber [origin [i]]* and update the residual edge-tokens accordingly.

Negative Cluster Firing: This is similar to the positive cluster firing pulse except that each node will examine the outgoing edges instead of incoming edges, and perform negative firing instead of positive firing.

Note that the above two pulses will not be executed concurrently.

Determining New Cluster: The purpose of this phase is to determine new fundamental clusters and determine their weights. There are two pulses in this phase.

In the first pulse, each node i with *num_tree_edges [i]* = 1 will do the following.

1. Write *clusterWeight [father [i]]* = *clusterWeight [father [i]]* + *clusterWeight [i]*.
2. Decrement by 1 *num_tree_edges [i]*.
3. Decrement by 1 *num_tree_edges [father[i]]*.
4. Write *origin [i]* = *father [i]*.

During the second pulse only nodes with *num_tree_edges [i]* = 0 will be active. Each such node i will write

$$origin\ [i] = origin\ [origin\ [i]]$$

Thus at the end of this pulse each node will have the *origin* of the fundamental cluster it is in as well as the weight of this cluster.

6.7 Avoidance of Cycling

We have incorporated in the Concurrent pivot phase, Bland's anti-cycling rule [2] to avoid occurrence of cycling. The details of the implementation of this anti-cycling strategy are omitted here for lack of space.

7 Experimental Results

The CBDS method has been implemented on the BBN Butterfly machine. The algorithm has been tested on several large graphs generated using NETGEN [6] program. To estimate parallel speed-ups, the algorithm has been tested for different numbers of processors. These results and those for the Modified Network Dual Simplex (MNDS) method of [9] are tabulated in Tables 1-5 where an iteration refers to the number of times the concurrent pivot step is performed. Our main purpose in testing is to demonstrate the scalability of these algorithms. So, the shared memory and processes created are distributed randomly among the hardware processors. Therefore, the inter-process communication is large and varies for each run and configuration. This is the reason for the large execution times of these algorithms. We notice that initially

the computer times decrease as the number of processors used increases and then stabilizes after reaching a maximum. Also, CBDS has better execution times and speed-ups as the size of the graphs increases.

8 References

[1] Chalasani, R.P., K. Thulasiraman, and M.A. Comeau, "Integrated VLSI Layout Compaction and Wire Balancing on a Shared Memory Multiprocessor: Evaluation of a Parallel Algorithm", ISPAN 94, Japan, 49-56, 1994.

[2] Chvatal, V., *Linear Programming,* Freeman Company, Potomac, Maryland, 1983.

[3] Comeau, M.A. and K. Thulasiraman, "Structure of the Submarking Reachability Problem and Network Programming," *IEEE Trans. Circuits and Systems,* Vol. CAS-35, 89-100, 1988.

[4] Comeau, M.A., K. Thulasiraman and K.B. Lakshmanan, "An Efficient Asynchronous Distributed Protocol to Test Feasibility of the Dual Transshipment Problem," Proc. Allerton Conf. on Communication, Control and Computer, Univ. of Illinois, Urbana, 634-640, Sept. 1987.

[5] Goldberg, A.V., "Efficient Graph Algorithms for Sequential and Parallel Complexity," *Ph.D. Thesis,* Lab. for Computer Science, M.I.T., Cambridge, MA, 1987.

[6] Klingman, D., A. Napier and J. Stutz, "NETGEN: A Program for Generating Large Scale Capacitated Assignment, Transportation and Minimum Cost Flow Problems," *Management Science*, 20, 814-821, 1974.

[7] Lengauer, T., *Combinatorial Algorithms for Integrated Circuit Layout,* John Wiley & Sons, England, 1990.

[8] Peters, J., "The Network Simplex Method on a Multiprocessor," *Networks,* Vol. 20, No. 7, 845-859, 1990.

[9] Thulasiraman, K., R.P. Chalasani and M.A. Comeau, "Parallel Network Dual Simplex Method on a Shared Memory Multiprocessor," *Proc. 5th IEEE Symp. on Parallel and Distributed Processing,* Dallas, 408-415, 1993.

[10] Thulasiraman, K., R.P. Chalasani, P. Thulasiraman and M.A. Comeau, "Parallel Network Primal-Dual Method on a Shared Memory Multiprocessor and A Unified Approach to VLSI Layout Compaction and Wire Balancing," *Proc. IEEE VLSI Design '93,* Bombay, India, 242-245, Jan. 1993.

[11] Thulasiraman, K. and M.A. Comeau, "Maximum-Weight Markings in Marked Graphs: Algorithms and Interpretations Based on the Simplex Method," *IEEE Transactions on Circuits and Systems,* Vol. CAS-34, No. 12, 1535-1545, December 1987.

[12] Thulasiraman, K. and M.N.S. Swamy, *Graphs: Theory and Algorithms,* Wiley-InterScience, New York, 1992.

[13] Yoshimura, T., "A Graph Theoretical Compaction Algorithm," *Proceedings of International Symposium on Circuits and Systems,* ISCAS 85, 1455-1458, 1985.

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	1502.96	1.0	1068.46	1.0
2	784.19	1.9	554.06	1.9
4	425.92	3.5	302.05	3.5
7	268.51	5.6	196.64	5.4
10	187.34	8.0	168.44	6.3
12	211.36	7.1	156.83	6.8
14	193.97	7.7	116.13	9.2
Iterations	7		4	

Table 1: Graph with 500 nodes and 12000 edges

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	25542.71	1.0	11360.06	1.0
2	12920.70	2.0	5978.98	1.9
4	5896.98	4.3	3005.84	3.8
7	3436.68	7.4	1829.64	6.2
10	2191.07	11.7	1231.60	9.2
13	2804.87	9.1	1107.76	10.3
15	1812.91	14.1	962.72	11.8
Iterations	5		3	

Table 2: Graph with 1200 nodes and 35000 edges

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	37022.75	1.0	40667.69	1.0
2	18697.69	2.0	20888.46	1.9
4	9385.40	3.9	10877.06	3.7
7	5113.30	7.2	5355.38	7.6
10	4181.29	8.9	3973.10	10.2
13	3264.70	11.3	2961.82	12.5
15	3895.15	9.5	2826.16	13.1
Iterations	3		3	

Table 3: Graph with 2000 nodes and 30000 edges