Parallel Network Dual Simplex Method on a Shared Memory Multiprocessor

K. Thulasiraman, Raghu P. Chalasani and M.A. Comeau

Concordia University, Montreal

Abstract

*We Present a parallel algorithm for solving the **dual transshipment problem** [1]. The traditional dual simplex method does not offer much scope for parallelization, because it moves from one basic feasible solution to another, performing one pivot operation at a time. We present a new method called **Modified Network Dual Simplex** method which uses concurrent pivots. This departure from the traditional LP approach raises several issues such as the need to convert a non-basic feasible solution to a basic feasible solution. We present our strategies to handle these issues as well as the corresponding parallel algorithms. We also present results of testing this algorithm on large graphs to solve the integrated layout compaction and wire balancing problem.*

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.

The **transshipment problem** which can be formulated as a linear programming problem is a general class of network optimization problem [1]. 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 [6], [11]. There are two basic approaches to the transshipment problem - the **network simplex method** and the **network primal dual method**. Goldberg [4] presented a variant of the primal-dual method called the ε-relaxation method. References to other ε-relaxation methods may also be found in [4]. 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 [7] has presented a parallel implementation of the network simplex method. More recently, we have discussed in [8] a parallel implementation of the network primal-dual method. Goldberg's algorithms in [4] are also suitable for parallel implementation.

In this paper, we describe development of a parallel algorithm for the dual transshipment problem and present results in the context of an application, namely, the VLSI layout compaction and wire balancing problem.

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. 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 **fun-**

damental cutset with respect to the edge e of T.

The notion of firing [2], [9] will be used extensively in the development of our algorithm in this paper. **Positive firing of a node** i, x times 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 of a node** i, x times 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). A subset of nodes will be called a **cluster**. 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 [2], [9]. Hence, unless otherwise stated, positive firing will be referred to simply as **firing**.

The **token of a directed path** will refer to the sum of the tokens of the edges on this path. The **token of a directed circuit** is defined similarly. d_{ij} will denote the token of a minimum-token directed path from node i to j. A minimum-token path from i to j will be referred to as a **shortest path from i to j**. For definitions of other standard graph-theoretic terms, [10] may be referred to.

3 The Dual Transshipment Problem

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 [10] and A^t be its transpose. The dual transshipment problem is a linear program defined as follows:

Maximize: W Y
subject to

$$A^tY \geq -M_0 \quad (1)$$

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

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. WY 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 $j \in \overline{S}$. Then S and $\overline{S}$ will be 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 [1] 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 [1]. 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 $\overline{S}$. To avoid cycling, Bland's anti-cycling rule [1] can be used in step (2) while selecting a pivot operation. For all details relating to the simplex method, [1] may be consulted.

4 Parallel Network Dual Simplex Method

As can be seen from the outline of the dual simplex method, this method moves from one basic feasible solution to another, performing one pivot operation at a time. Any parallel implementation of this method will focus on the parallelization of the pivot operation. But during a pivot operation only a small subgraph of the given graph will be involved. Thus such an approach does not offer much scope for achieving good speed up.

We resolve the above difficulty by permitting concurrent pivot operations. But at the end of concurrent pivots, the resulting solution may not be basic. So, we need an algorithm to generate a basic feasible solution from a given feasible solution. But while doing so we should ensure that the objective value WY never decreases. Our method to be called **Modified Network Dual Simplex Method** takes care of these considerations. An outline of this method is as follows.

Modified Network Dual Simplex Method

S1: Test feasibility of the DTP. If feasible, construct a feasible solution.

S2: Given a feasible solution Y, construct a basic feasible solution Y′ with WY′ ≥ WY.

S3: Check the optimality of the basic feasible solution Y′ obtained in S2. If it is optimal, the algorithm terminates. Otherwise, perform concurrent pivot operations starting from Y′. (This involves selecting the fundamental clusters defined by the basic feasible tree T corresponding to Y′ and firing them in an appropriate manner).

S4: Repeat S2.

We now proceed to discuss each step in the above algorithm and describe the details of a parallel algorithm. In our model of 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. All our algorithms have been implemented on the Shared memory BBN Butterfly machine.

4.1 Feasibility Testing

It can be shown that the DTP is feasible if the graph has no directed circuit of negative token. Let $\sigma_i = \text{Max}\ \{0, -\min\ \{d_{ij}\}\}$. Then we have the following result [2].

Theorem 1

The vector $Y = (\sigma_1, ..., \sigma_n)$ is a feasible solution of the DTP if the graph G has no directed circuit of negative token. ❑

In [3], we have presented a distributed algorithm to test feasibility of a given DTP. This algorithm can easily be adapted for shared memory implementation. For proof of correctness and other details of this algorithm [3] may be consulted.

This method does not require constructing an auxiliary network to test feasibility. This is an attractive feature from the point of view of designing distributed/parallel implementations. Also, we need no explicit synchronizer mechanism unlike in our distributed implementation discussed in [3]. The inherent synchronization available when one spawns different processes in the BBN itself serves as a synchronizer.

4.2 Constructing a Basic Feasible Solution

Given a feasible solution Y of the DTP, we would like to construct a basic feasible solution Y′ such that WY′ ≥ WY. To do so we proceed as follows.

First, we fire the non-negative weight nodes concurrently as much as possible. We then repeat these concurrent firings until no further firings are possible. These firings will not decrease the value of the objective WY.

We now illustrate a difficulty that we may encounter if we perform the node firings as above. Consider the graph shown in Fig. 1. During the first pulse of concurrent firings, all the nodes will be fired exactly once. After twenty pulses of these firings, the residual tokens will be as shown in Fig. 1(b). After the 23rd pulse, the residual tokens will be as shown in Fig. 1(c). The total number of firings of each node is also given in this figure.

On the other hand, suppose we first determine the okens of the shortest paths from node a to all other nodes. We find that these tokens are precisely the number of times the different nodes could be fired as in Fig 1(c). This is not an accident. In fact we can prove the following.

Theorem 2

If there exists a directed path from i to j of token x, then the node i can be fired at most x times. ❑

So computing first the tokens of shortest paths and then firing would save a large number of pulses. (Note that, for this example, shortest path computations will take only 5 pulses). In our algorithm we employ this strategy. Note that since we are interested in firing only positive weight nodes, we need to compute for each positive node j, the token of a shortest path from a non-positive weight node.

Note that some of the residual edge tokens corresponding to a feasible solution may be equal to zero. But a node i with an incoming edge (j, i) of zero token cannot be fired at all, if node j is negative-weighted. In such cases, we should group nodes i and j into one cluster and consider firing this group. So, our strategy is to first partition the node set V into subsets $S_1, ..., S_k$ such that each S_i with $|S_i| > 1$ has at least one non-positive weight node and all nodes reachable from this node from a directed path of zero residual token are also in S_i. If a node is reachable from more than one non-positive weight node, then all such non-positive nodes will also be in S_i. Firing these clusters of nodes is then considered.

Summarizing, our approach to construct a basic feasible solution from a given feasible solution involves repeated applications of the following phases.

1. Clustering.
2. Contraction. (Note that if we find, after contraction, that the nodes of G have joined to form one single cluster, then we have reached a basic feasible solution. In this case, the phases listed below will not be necessary. See Section 4.2.3.)
3. Shortest path computation and firing.

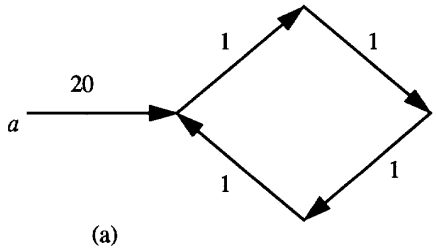


Figure 1. Illustration of a Problem

4.2.1 Clustering

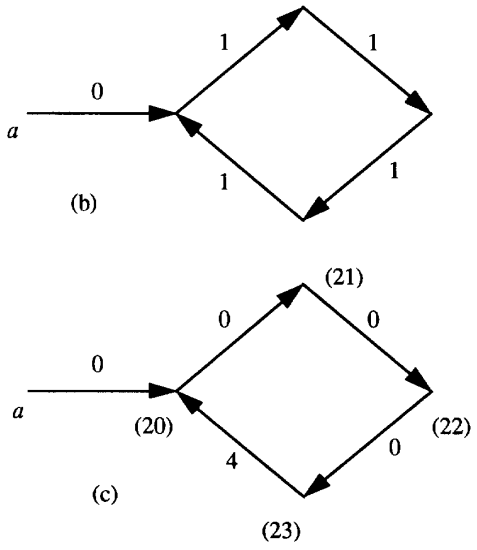
Clustering involves finding for each non-positive weight node i, the set of all nodes reachable from i by directed paths with zero residual tokens. If a positive weighted node is reachable from more than one non-positive weighted node, then the union of the corresponding sets will represent the cluster containing node j. The **root of a cluster** is the least-numbered non-positive weighted node in that cluster. In our description of the clustering process, we use the following data structures.

source is an array of integers. The variable *source [i]* at the end of this phase represents the least numbered non-positive node in the cluster in which node i is present. *source [i]* is initialized to NOTHING if the weight of the node i is positive, otherwise it is initialized to i. *zeroNodes* is an array of sets. *zeroNodes [i]* contains all zero outnodes at node i. *job_done* is an array of boolean. In the beginning, each *job_done [i]* is initialized to FALSE. As each node propagates its *source* to all of its zero-outnodes, it will change its *job_done [i]* to TRUE. Note that node j is a zero-outnode at node i if node j is an outnode at node i and the residual token of (i, j) is zero.

The clustering phase has three subphases.

Subphase 1: For the first pulse of this subphase, variables are initialized as described above. During subsequent pulses each node i performs the following actions.

If *source [i]* = NOTHING, no action is taken. Otherwise, if *job_done [i]* = FALSE, then node i sets *job_done [i]* = TRUE and performs the following for each zero outnode j.



1. If *source [j]* = NOTHING, node *i* writes *source [j]* = *source [i]*.
2. If *source [j]* ≠ NOTHING. then node *i* will do the following operations.
   (a) Node *i* examines the source variables in the order *source [i]*, *source [source [i]]*, ..., *source [source [... [source [i] ...]]*and picks the first node *k* in this sequence for which *source [k]* = *k*.
   (b) Node *i* examines the source variables in the order *source [j]*, *source [source [j]]*, ..., *source [source [... [source [j] ...]]*and picks the first node *l* in this sequence for which *source [l]* = *l*.
   (c) If *l* < *k*, then node *i* writes *source [k]* = *l* and *source [i]* = *l*; otherwise it writes *source [l]* = *k*.

The above pulses of operations are repeated until *job_done (i)* = TRUE for every node *i* or no action is taken by any processor during a pulse.

Note that at the end of the first subphase each non-positive node *i* will have *source [i]* ≤ *i*. It is likely that certain positive nodes have their *source* variables equal to NOTHING, unchanged from their initial values. For such nodes we set *source [i]* = *i* before initiating subphase 2.

Subphase 2: In this subphase only non-positive nodes will be active. During this subphase, each node *i* performs the following actions.

Node *i* examines the source variables in the order *source [i]*, *source [source [i]]*, ..., *source [source [... [source [i] ...]]*and picks the first node *k* in this sequence for which *source [k]* = *k*. Node *i* then writes *source [i]* = *k*.

We can show that at the end of the second subphase, *source [i]* for each non-positive node will contain the least numbered non-positive node in its cluster.

Subphase 3: In this subphase, each positive node *i* will write *source [i]* = *source [source [i]]*.

It can be shown that, at the end of the third subphase, all nodes with the same value for their *source* variables represent the members of a cluster. This *source* value will also give the least-numbered non-positive node in that cluster if the cluster has cardinality greater than 1. Thus, *source [i]* is in fact the root of the cluster containing node *i*.

4.2.2 Contraction

Let S_1, ..., S_k be the clusters formed by the clustering phase. Recall that *source [i]* denotes the root of cluster S_i. Now we wish to contract all the nodes in each cluster and construct implicitly a graph G′ in which each node represents a cluster. Also in this graph, there will be at most one edge directed from *source [i]* to any *source [j]*, j ≠ i. The residual token of such an edge will be the minimum of the tokens of all edges in G directed from a node in S_i to a node in S_j. The weight of each cluster and the corresponding node in G′ will be the sum of weights of all the nodes in that cluster.

Our algorithm for the contraction phase assumes that the following data structures are available at each node. *clusterWeight* is 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 *i* containing node *i*. *outNodes* is an array of sets. *outNodes [i]* is a set containing all the nodes *j* such that (*i*, *j*) is an edge in the given graph G. *activeOutNodes* is an array of sets. *activeOutNodes [i]* is a set containing all the nodes *j* such that (*i*, *j*) is an edge in the current contracted graph G′. This variable is initialized to *outNodes [i]*. *tokens* is a 2-dimensional array of integers. *tokens [i] [j]* represents the value of edge-token for the edge *(i, j)* in the given graph G. If this edge is not in G, *tokens [i] [j]* contains a special value INFINITY. *activeTokens* is a 2-dimensional array of integers. *activeTokens [i] [j]* represents the value of the residual edge-token for the edge *(i, j)* in the current contracted graph G′. If this edge is not in G′, *activeTokens [i] [j]* contains a special value INFINITY outside the range of edge tokens. This variable is initialized to *tokens [i] [j]*. *lock* is an array of integers. Each *lock [i]* is initialized to zero.

Now we proceed to the details of our algorithm to construct the contracted graph G′ from G.

In G′, each cluster will be represented by its root. The weight of a root will be the weight of the corresponding cluster. To compute this weight, each node *i*, if it is not a root, will add its weight to its root's weight. The value of the *source* variable will also tell if a node is a root node or it belongs to a cluster, or simply if it is a node by itself. If *source [i]* is equal to *i*, then the node *i* is the root of the cluster containing node *i*. If *source [i]* is equal to NOTHING (a special value), then the node *i* belongs to a cluster of cardinality equal to unity. This could be the case for some positive nodes especially after the Feasibility phase. If *source [i]* is not equal to *i* and is also not equal to NOTHING, then the node *i* belongs to the cluster containing *source [i]*. In this case, node *i* should add *clusterWeight [i]* to the *clusterWeight [source [i]]*.

That is, node *i* writes

clusterWeight[source[i]] = clusterWeight [source[i]] + clusterWeight [i], if *source [i]* ≠ *i*.

Since there may be more than one node trying to access the source's clusterWeight variable at the same time, they lock the clusterWeight variable of the source

before they update it. After the update, the node will unlock it so that another node can update it. Note the locking and unlocking of each source's clusterWeight variable can be done independently.

Next the set of *activeOutNodes* as well as residual tokens of edges (*activeTokens*) in G´ have to be calculated. This is accomplished in two steps.

First, each node i will process its *activeOutNodes [i]* set one by one. For example, assume node i is processing *activeOutNodes [i]*. Suppose k is the source for the node i. Further, suppose that there is an edge from node i to j, and therefore j is an out-node at i and l is the source of the cluster in which j is present. Then node i takes the following actions.

1. Remove j from *activeOutNodes [i]*.
2. If $k \neq l$, add l to *activeOutNodes [i]*.
3. If *activeTokens [i] [l] > activeTokens [i] [j]*, then write *activeTokens [i] [l] = activeTokens [i] [j]*.

In the second step, each node will process the edges in *activeOutNodes [i]* so that root's *activeTokens* set will contain only the most constraining edges, and then combine its *activeOutNodes* set with that of the root. That is, it will calculate *activeOutNodes[i]* $\cup$ *activeOutNodes [source [i]]*.

Note that there may be conflicts among nodes when they try accessing their source's data structures. Therefore locking and unlocking of a source's data structure is necessary in this step. But the data structure of each source can be locked and unlocked independent of each other.

At the end of the second step, all nodes except the roots will become inactive, thus making contraction of the graph complete.

4.2.3 Shortest Path Computations and Firings

We now have a contracted graph G´ in which each node represents a cluster of nodes of the given graph G. We also have for each node in G´ its weight (the weight of the corresponding cluster) and for each edge its residual token.

We now need to find for each positive node i in G´ a shortest path from a non-positive weight node. The token of such a path specifies the number of times node i could be fired without resulting in any negative residual token. These shortest path computations can be done in parallel by generalizing Bellman-Ford-Moore's single-source shortest path algorithm to the multiple-source case. We refer to [5] for details of such an algorithm. It should be noted that the graph G´ under consideration will have no negative residual tokens and so successful termination of this shortest path algorithm will occur in less than n pulses. Also, it should be noted that if a positive weight node i is not reachable from any non-positive node, then the shortest path algorithm will terminate with *distance (i)* = ∞. This indicates unboundedness of the DTP because of the node i can be fired an unbounded number of times increasing the value of WY to an unbounded value.

This completes our discussion of the three phases in our algorithm to construct a basic feasible solution.

Summarizing, our algorithm to construct a basic feasible solution starting from a given feasible solution Y is as follows.

1. Perform on G, the following sequence of operations.
   (a) Clustering.
   (b) Contraction.
   (c) Shortest path computation and firings.
2. If clustering results in a graph G´ which consists of exactly one node, proceed to step (3), otherwise let G = G´ and repeat step 1.
3. (At this point, we have a spanning subgraph of G in which residual tokens of all edges are equal to zero and so the node firing numbers represent a basic feasible solution.) Build a 0-token spanning tree of G. This tree is a required basic feasible tree.

A 0-token spanning tree as required in step (3) above can be easily built using a parallel depth-first search of the subgraph of zero-token edges referred to in step (3), if at each node i we maintain the set consisting of the nodes adjacent to i. Maintaining such a set has not been necessary for any of the algorithms discussed thus far in this section. Note that we have so far used only the sets *outNodes [i]*. We have designed a parallel algorithm to build the required spanning tree using only these sets. To conserve space, the details of this algorithm are not included.

4.3 Concurrent Pivots

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] = node-*

Weight [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]] $= f_i$,
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.

4.4 Avoidance of Cycling

We have incorporated in the Concurrent pivot phase, Bland's anti-cycling rule [1] to avoid occurrence of cycling. The details of the implementation of this anti-cycling strategy are omitted here for lack of space.

5 Experimental Results

As we mentioned in Section I, VLSI layout compaction and wire balancing (or wire length minimization) is an area of application of our work. Wire balancing problem can be formulated as a dual transshipment problem [6], [11]. In this formulation, Y will denote the vector of positions of nodes (cells) and each node weight w_i will denote the relative change in cost due to unit change in y_i. Each $(-m_{ij})$ will denote the minimum spacing required between nodes i and j.

In layout compaction, we are required to place the nodes (cells) in a layout so that the chip area is minimum. We have shown in [8] that this, in fact, is equivalent to testing feasibility of a DTP. That is, testing if the constraints

$$A^t Y \geq -M_0$$

$$Y \geq 0$$

have a feasible solution. Thus our algorithm described in Section 4.1. achieves layout compaction in parallel.

After placing the nodes at positions specified by the values of σ_i's (See Theorem 1), positions of some of the nodes can be adjusted (without violating feasibility or increasing chip area) if one is interested in minimizing wire length after compacting the chip. This problem called the **integrated layout compaction and wire balancing** problem, involves the following steps:

1. Find the feasible vector Y as specified in Theorem 1.
2. For certain nodes, fix their values as specified by the Y obtained in step 1. (These nodes are in fact those which lie on longest paths in G.) Then solve the DTP.

Our parallel algorithm for the DTP can thus be adapted to solve the different problems related to compaction and wire balancing mentioned above.

We have adapted our algorithm for the integrated lay-

out compaction and wire balancing problem and tested this on several graphs (representing layouts of industrial designs received from Cadence Design Systems, Inc.). To estimate parallel speed-ups, the algorithm has been tested for different numbers of processors. The test results are given in Tables 1-3. We notice that the computer times initially decrease as the number p of processors used increases and then stabilizes after reaching a minimum.

Acknowledgement

We would like to thank Ravi Varadarajan of Cadence Design Systems, San Jose for providing us with test graphs as well as giving insight into the integrated layout compaction and wire balancing problem.

We would also like to thank Paul Gaudet of BBN Systems, Inc. for giving us access to the BBN Butterfly machine and helping us through whenever we had problems while using this machine.

6 References

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

[2] 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.

[3] 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.

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

[5] Lakshmanan, K.B., K. Thulasiraman and M.A. Comeau, "An Efficient Distributed Protocol for Finding Shortest Paths in Networks with Negative Weights," *IEEE Transactions on Software Engineering,* Vol. 15, No. 5, 639-644, May 1989.

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

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

[8] 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. VLSI Design '93,* Bombay, India, 242-245, Jan. 1993.

[9] 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.

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

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

Table 1: Graph with 149 nodes

Processors p	Time in Seconds	Parallel Speed up
1	22.97	1.0
4	7.63	3.0
6	5.98	3.8
8	4.30	5.3
10	3.90	5.9
13	3.93	5.9
14	3.84	6.0
15	3.65	6.3

Table 2: Graph with 157 nodes

Processors p	Time in Seconds	Parallel Speed up
1	18.29	1.0
4	5.60	3.3
6	4.45	4.1
8	3.06	6.0
10	3.01	6.1
11	2.97	6.2
16	2.76	6.6
17	2.79	6.6

Table 3: Graph with 2087 nodes

Processors p	Time in Seconds	Parallel Speed up
1	10572.73	1.0
2	5111.41	2.1
4	2600.39	4.1
8	1631.72	6.5
12	1174.22	9.0
15	1010.37	10.5
17	996.62	10.6
20	948.61	11.1