PARALLEL NETWORK OPTIMIZATION ON A SHARED MEMORY MULTIPROCESSOR AND APPLICATION IN VLSI LAYOUT COMPACTION AND WIRE BALANCING

RAGHU PRASAD CHALASANI

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the requirements

for the degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

March 1994

ISBN 0-315-90832-7

ABSTRACT

Parallel Network Optimization on a Shared Memory Multiprocessor and Application in VLSI Layout Compaction and Wire Balancing

Raghu Prasad Chalasani, Ph.D.
Concordia University, 1994.

Network optimization refers to the general class of optimization problems defined on graphs and networks. The transshipment and the dual transshipment problems generalize several of the network optimization problems. They can be used to formulate and solve a large class of industrial and engineering problems. As the complexity and sizes of these problems increase, there is a growing demand for more computing power. Parallel computing is one of the ways to meet this demand. This has motivated our work in this thesis on parallel network optimization.

Towards this end, we discuss the design and implementations of three parallel algorithms - one algorithm for the transshipment problem and two algorithms for the dual transshipment problem. We also consider an application of these algorithms in solving VLSI layout compaction and wire balancing problems.

Three basic network optimization algorithms serve as building blocks for our parallel algorithms. They are: Algorithm FEASIBLE to test feasibility of the dual transshipment problem, Algorithm SHORTEST-PATH to compute shortest paths and Algorithm MAX-FLOW to compute a maximum flow in a network. Our parallel algorithms for the dual transshipment problem extensively employ the notions of node/cluster firings and results from the theory of marked graphs.

Our parallel network primal dual method for the transshipment problem is based on the traditional network primal dual method. We show that Algorithm FEASIBLE can be adapted to initialize primal dual method. This obviates the need for constructing an

auxiliary network. This is an attrative feature from the point of view of a parallel implementation.

Two new approaches - MNDS (Modified Network Dual Simplex) and CBDS (Cluster-Based Dual Simplex) - are developed for solving the dual transshipment problem. In both these approaches, we do not need to move from one basic solution to another. Constructing efficiently a basic feasible solution starting from a feasible solution and performing efficiently concurrent pivots without destroying feasibility are the distinguishing features of these two approaches. They differ from one another significantly in the way they construct a basic feasible solution. A new characterisation of the structure of the optimum solution of the dual transshipment problem is the basis of the CBDS method.

Finally, we show that the wire balancing problem in VLSI physical design can be formulated as a dual transshipment problem and that layout compaction is a special instance of this problem. In fact, layout compaction can be achieved using Algorithm FEASIBLE. We also introduce the integrated layout compaction and wire balancing problem and present a unified formulation of this problem using the dual transshipment problem.

A comparative experimental evaluation of our parallel algorithms is also provided.

ACKNOWLEDGEMENTS

I am grateful to many people that provided help and inspiration for this work. First, I would like to thank my thesis supervisor Prof. K. Thulasiraman. The technical guidance I received and the long discussions we had through out the span of this research were truly valuable.

I like to express my appreciation to Dr. M.A. Comeau for his helpful and encouraging suggestions.

I am thankful to Ms. Parimala Thulasiraman for helping me with Primal Dual algorithm.

I would also like to thank Mr. Ravi Varadarajan of Cadence Design Systems, Inc. for providing me the test graphs for layout compaction and wire balancing problems as well as giving insight into the integrated layout compaction and wire balancing problem.

I would like to thank Dr. C. Tropper for providing access to BBN Butterfly parallel computer.

Special thanks are due to Mr. Nicky Ayoub for patiently helping me through whenever I had systems problems.

Finally, I would like to thank my family and friends for their help and support in making this thesis a reality.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

CBDS	Cluster-Based Dual Simplex
BFM	Bellman-Ford-Moore
DTP	Dual Transshipment Problem
LP	Linear Program
MNDS	Modified Network Dual Simplex
VLSI	Very Large Scale Integration
N	A Network
G (V, E)	A Connected Directed Graph
V	Node Set
$v_1, v_2, ..., v_n$	Nodes of the graph
E	Edge Set
$e_1, e_2, ..., e_n$	Edges of the graph
(i, j)	Edge directed from node i to node j
A	Incidence matrix of the graph G
A^*	– A
c_{ij}	Capacity of edge (i, j)
C	Column vector of edge capacities c_{ij}
m_{ij}	Token of edge (i, j)
M	Row vector of edge tokens m_{ij}
W	Column vector of node supplies or demands
$x_{ij}, f_{ij}, f(i, j)$	Flow on the edge (i, j)
X	Column vector of edge flows x_{ij}
y_i	Firing number of node i
Y	Column vector of node firing numbers y_i
d_{ij}, d(i, j)	Length of a shortest path from node i to node j
$val\ (f)$	Value of flow f
$e_f(i)$	Excess flow at node i
E_f	Residual graph
$\in$	Set Member of
$\cup$	Set Union
$\cap$	Set Intersection
$\supseteq$	Superset
$\subseteq$	Subset
–	Set Difference
| S |	Cardinality of Set S
$\overline{S}$	Complement of Set S

CHAPTER 1

INTRODUCTION

In the last decade or so, there has been an intensive interest in parallel processing research. This can be attributed to the fact that rapid advances in microelectronics technology have made available high-performance parallel computers at relatively lower cost. Fast hardware is essential for many applications, such as those that require real-time computations. Furthermore, by interconnecting computers in an organization and using parallel/distributed algorithms, one can solve efficiently several classes of problems whose complexity is beyond the ability of any one computer in terms of memory and computing requirements. Such algorithms also result in effective utilization of computing hardware.

Parallel machines with tens of thousands of processors are already available commercially, and machines with millions of processors can be built with today's technology. Significant progress has been achieved in the application of parallel computing for numerical problems. Such is not the case for problems which involve non-numerical computations. Perhaps the reason is that until recently, most computationally intensive applications - aerospace, signal processing, physics etc. - have all been mainly of a numerical nature. The design and implementation of parallel algorithms for non-numerical problems is also not well understood.

The rapidly increasing complexity of VLSI circuits puts pressure on VLSI CAD tools from two directions. First, the amount of data to be processed has risen dramatically. Second, VLSI designers have come to rely more heavily on CAD tools and demand higher performance. This demand for speed up of the VLSI design process calls for an application of parallel computing technology. In recent years, parallel algorithms for certain phases (in particular, VLSI physical design) in the VLSI design process have been reported (See Section 1.2.2). Computations which one encounters in several steps of VLSI

physical design are essentially non-numerical in nature. Almost all problems which arise in VLSI physical design (layout: floor-planning, placement, routing, compaction, etc) can be formulated in terms of graph and network optimization problems. Besides VLSI physical design problems, a number of other problems which arise in VLSI design can also be formulated as network optimization problems. This motivated us to look into the issue of designing efficient parallel algorithms for network optimization problems.

The transshipment problem [39] which can be formulated as a linear programming problem is a general class of network optimization problems. Several network optimization problems such as constructing shortest paths, finding a maximum flow, constructing a min-cost spanning tree, finding matchings in networks etc. are special cases of the transshipment problem. These network optimization problems occur routinely as building blocks in designing efficient algorithms for more complex problems. The need for parallel algorithms for the transshipment problem and its variants is apparent when one sees the range of large engineering applications that can be modeled as a transshipment problem or its dual. This is indeed the case in VLSI physical design research.

Motivated by the above considerations, this thesis addresses issues in design and efficient implementation of parallel algorithms for the transshipment problem and its dual as well as their application in VLSI physical design and, in particular, in VLSI layout compaction and wire balancing problems.

1.1 Models of Parallel Computations

A detailed discussion of different models of parallel computations and related issues may be found in [6], [19]. The most widely used taxonomy to characterize parallel computations is the one proposed by Flynn [19]. He identifies four classes: Single Instruction, Single Data stream (SISD), Single Instruction, Multiple Data stream (SIMD), Multiple Instruction, Single Data stream (MISD), and Multiple Instruction, Multiple Data

stream (MIMD). The model which one uses would depend on the structure of the computations involved. In the following sections, the above four models of parallel computations are briefly discussed. More details of these models may be found in [6] and [19].

1.1.1 SISD Computers

In this model, at each step during a computation the control unit emits one instruction that operates on a datum obtained from the memory unit. Such an instruction may tell the processor, for example, to perform some arithmetic or logic operation on the datum and then put it back in memory.

The overwhelming majority of computers today adhere to this model invented by John von Neumann and his collaborators in the late 1940s. An algorithm for a computer in this class is said to be *sequential (or serial).*

1.1.2 MISD Computers

Each of N processors in the class of MISD computers has its own control unit, but all of them share a common memory unit where data reside. There are N streams of instructions and one stream of data. At each step, one datum received from memory is operated upon by all the processors simultaneously, each according to the instruction it receives from its control. Thus, parallelism is achieved by letting the processors do different things at the same time on the same datum. This class of computers lends itself naturally to those computations requiring an input to be subjected to several operations, each receiving the input in its original form.

1.1.3 SIMD Computers

In this class, a parallel computer consists of N identical processors. Each of the N processors possesses its own local memory where it can store both programs and data. The processors operate synchronously: At each step, all processors execute the same instruction, each on a different datum.

Sometimes, it is desirable for the processors to be able to communicate among themselves during the computation in order to exchange data or intermediate results. This can be achieved in two ways, giving rise to two subclasses: SIMD computers where communication is through a *shared memory* and those where it is done via an *interconnection network*.

1.1.4 MIMD Computers

Here there are N processors, N streams of instructions and N streams of data. Each processor operates under the control of an instruction stream issued by its control unit. Processors typically operate asynchronously, though some algorithms possess structures amenable for synchronous implementation. Communication between processors is through a shared memory or an interconnection network. MIMD computers sharing a common memory are often referred to as multiprocessors (or tightly coupled machines) while those with an interconnection network are known as multicomputers (or loosely coupled machines). When a shared memory is involved, issues such as concurrent memory access conflicts need to be studied and resolved. A review of different popular interconnection networks may be found in [6], [19].

The MIMD model of parallel computation is the most general and powerful model possible. Computers in this class are used to solve in parallel those problems that lack the regular structure required by the SIMD model. This generality does not come for free: Asynchronous algorithms are difficult to design, evaluate, and implement. In order to appreciate the complexity involved in programming MIMD computers, it is important to distinguish between the notion of a process and that of a processor. An asynchronous algorithm is a collection of processes some or all of which are executed simultaneously on a number of available processors. Initially, all processors are free. The parallel algorithm starts its execution on an arbitrarily chosen processor. Shortly thereafter it creates a number of computational tasks, or processes, to be performed. A process thus corresponds to a

section of the algorithm: There may be several processes associated with the same algorithm section, each with a different parameter.

Once a process is created, it must be executed on a processor. If a free processor is available, the process is assigned to the processor that performs the computations specified by the process. Otherwise (if no free processor is available), the process is queued and waits for a processor to become free.

When a processor completes execution of a process, it becomes free. If a process is waiting to be executed, then it can be assigned to the processor just freed. Otherwise (if no process is waiting), the processor is queued and waits for a process to be created.

The order in which processes are executed by processors can obey any policy that assigns priorities to processes. For example, processes can be executed in a first-in-first-out order. Also, the availability of a processor is sometimes not sufficient for the processor to be assigned a waiting process. An additional condition may have to be satisfied before the process starts. Similarly, if a processor has already been assigned a process and an unsatisfied condition is encountered during execution, then the processor is freed. When the condition for resumption of that process is later satisfied, a processor (not necessarily the original one) is assigned to it. These are but a few of the scheduling problems that characterize the programming of multiprocessors. Finding efficient solutions to these problems is of paramount importance if MIMD computers are to be efficiently used. Note that none of these scheduling problems arise on the less flexible but easier to program SIMD computers.

The MIMD (Multiple Instruction stream and Multiple Data stream) model seems to be the most appropriate one for the problems encountered in VLSI physical design. Therefore we use this model as the basis for the development of our algorithms.

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. The 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.

1.2 Review of Literature

1.2.1 Parallel Network Optimization

Kruskal [87] and Prim's [135] algorithms for the minimum cost tree problem, Dijkstra's [49] and Bellman-Ford-Moore's algorithms [18], [56], [109] for the shortest path problems, Ford-Fulkerson's algorithm [55] and its several variants (for example, [50], [102]) for the maximum flow problem are among the most fundamental algorithms in network optimization theory [164]. They have also served as the basis for designing corresponding distributed/parallel algorithms. Some of these distributed network algorithms may be found in [36], [61], [73] and [94]. Synchronizers have been designed for efficient implementation of asynchronous algorithms. Issues relating to the synchronizer design problem have been discussed in [93], [94] and [145].

A detailed discussion of the transshipment problem and algorithmic solutions may be found in [39], [138]. There are two basic approaches to this problem - the network simplex method and the primal-dual method. Goldberg [63] presented a variant of the primal-dual method called the ε-relaxation method. References to other ε-relaxation methods may also be found in [63]. The relaxation methods are primarily designed to obtain good com-

plexity results for the transshipment problem. Goldberg [63] and Goldberg and Tarjan [64] also presented a novel algorithm for the maximum flow problem. This algorithm for the maximum flow problem and the primal-dual approach for the transshipment problem and its variants are quite elegant and amenable to distributed implementation.

Stunkel [154] has reported results of both simplex and revised simplex implementations on an iPSC/2 hypercube. The methods are primarily row or column oriented and thus are more suited to large or medium-grained machines. Peters [131] has presented a parallel implementation of the network simplex method for solving the transshipment problem. Recently, an overview of parallel algorithms for solving discrete optimization problems has been provided in [65]. It also provides an extensive reference list of different parallel algorithms for discrete optimization problems. A distributed protocol for the network primal dual method and its implementation on a shared memory multiprocessor are discussed in [159]. A parallel algorithm for the transshipment problem based on the primal-dual method and a unified approach to layout compaction and wire balancing is presented in [161]. A parallel algorithm for the dual transshipment problem is discussed in [160]. This approach removes shortcomings of the dual simplex which made the latter unattractive for parallelization.

A general method for parallelizing Depth-First-Search (DFS) is presented in [54], [90], [91], [107]. This method employs stack splitting in conjunction with receiver initiated subtask distribution schemes. Each processor searches a disjoint part of the search space in a depth-first fashion. When a goal is found, all of them quit. Saletore and Kale [144] presented a parallel formulation in which nodes are assigned priorities and are expanded accordingly. By doing this, they were able to ensure that the nodes are expanded in the same order as the corresponding sequential formulation. Parallel DFS using sender initiated subtask distribution has been proposed by a number of researchers [53], [59], [133], [137], [148].

A number of load balancing schemes for DFS have been explored in [1], [9], [53], [59], [66], [90], [127], [133], [137], [144], [148]. Finkel and Manber [54] presented performance results for a number of problems such as the travelling salesman problem and Knights tour for the Crystal multicomputer developed at the University of Wisconsin. Monien and Vornberger [108] showed linear speedups on a network of transputers for a variety of combinatorial problems. Kumar et al [8], [90] have showed linear speedups for problems such as 15 puzzle and tautology verification for various architectures such as a 128 processor BBN Butterfly, 128 processor Intel Hypercube, a 1024 processor nCUBE2, and a 128 processor Symult 2010. Kumar et al [66], [90], [91], [113] have investigated the scalability and performance of many of these schemes for a variety of architectures such as hypercubes, meshes and networks of workstations.

Recent work has shown that SIMD architectures such as CM2 can also be used to implement parallel DFS algorithms effectively. Powley et al [134] and Mahanti and Daniels [101] presented parallel cost bounded DFS for solving the 15 puzzle problem on CM2. Powley's and Mahanti's formulations use different triggering and redistribution mechanisms. However, both schemes report similar results for the 15 puzzle. Karypis and Kumar [80] have presented a new load balancing technique which is shown to be highly scalable on SIMD architectures and is shown to be no worse than that of the best load balancing schemes on MIMD architectures.

Bixby [21] presented a parallel branch and cut algorithm for solving the symmetric traveling salesman problem. He also presented solutions of the LP relaxations of airline crew-scheduling models. Miller [105] presented parallel formulations of the best-first branch and bound technique for solving the asymmetric traveling salesman problem on heterogeneous network computer architectures. Roucairol [142] has presented parallel best-first branch and bound formulations for shared memory computers and used these to solve the Multiknapsack and Quadratic assignment problems. Pardalos and Crouse [123]

proposed a parallel formulation of the quadratic assignment problem based on best-first branch and bound search. Experimental results for parallel formulations of more general problems such as the quadratic 0-1 programming problem (using distributed and shared memory machines) are discussed in [124], [125].

Guibas, Kung and Thomson [68] have developed a systolic algorithm for the parenthesization problem. Karypis and Kumar [81] analyzed three different mappings of the systolic algorithm presented by Guibas et al [68] and experimentally evaluated them in the context of the matrix multiplication parenthesization problem on an nCUBE2. Huang et al [72] have presented an algorithm for solving a generalized parallel formulation of the dynamic programming (DP) problem on a CREW-PRAM. This formulation can be applied to a number of problems. DeMello et al [47] used vectorized formulations of DP for the Cray to solve optimal control problems. Lee et al [95] have used the divide and conquer strategy for parallelizing the DP algorithm for the 0/1 knapsack problem on a MIMD distributed memory computer. They demonstrated experimentally that it is possible to obtain linear speedup for large instances of the problem provided enough memory is available.

1.2.2 Parallel CAD for VLSI

In recent years, there has been an intensive research in the application of parallel computing technology to VLSI CAD problems. Parallel algorithms have been published for virtually every layout problem except for the integrated layout compaction and wire balancing problem even though it is a critical step in the final phase of the layout process. Two recent reviews of this research may be found in [11], [158]. An excellent tutorial on parallel algorithms for simulations may be found in [4]. Parallel placement has received extensive attention. Parallel routing and floorplan design have received some attention. A comprehensive treatment of parallel algorithms for VLSI physical design problems may be found in the forth coming book [12].

Research in parallel algorithms for placement has been primarily directed towards those based on simulated annealing [82]. The motivation for this stems from the success of the Timber-Wolf placement uniprocessor program which employs simulated annealing. A major limitation of this program is that it is very slow. So efforts have been directed towards speeding up simulated annealing through parallel processing. Banerjee, Jones and Sargent [13] have discussed parallel placement algorithms targeted to run on hypercube computers. Casotto and Sangiovanni-Vincentelli [31] have given a parallel placement algorithm for the connection machine.

Casotto et al [30], and Kravitz and Rutenbar [86] have discussed parallel placement algorithms and their implementations on shared-memory multiprocessors. Darema et al [45] have presented a parallel algorithm for placing a number of equal sized modules connected by multiterminal nets on a chip represented as a rectangular array. Their algorithm is based on simulated annealing on a shared memory multiprocessor. Natarajan and Kirkpatrick [118] have reported a different parallel simulated annealing algorithm on a shared memory multiprocessor using a decomposition approach. A method for standard cell placement based on a genetic algorithm has been proposed by Mohan and Mazumder [106].

Iosupovici, King and Breuer [74] have proposed a special purpose hardware architecture using a pipelined processor to rapidly evaluate alternative configurations of placements through placement interchanges among modules. A special purpose SIMD architecture for the above task has been proposed by Kumar and Patnaik [88]. Kumar and Sastry [89] have discussed a hardware accelerator for a module placement algorithm based on the Divide and Conquer paradigm. Kling and Banerjee [83] presented a parallel placement program designed to run on a network of loosely coupled multiprocessors such as workstations connected via ETHERNET. Rose et al [141] have discussed a parallel placement algorithm which improves upon simulated annealing. A parallel placement

algorithm based on pairwise interchange has been proposed by Sugiyama and Watanabe [155]. Ueda et al [167] have discussed a placement algorithm using adjacent pairwise interchange on a SIMD architecture with a two-dimensional processor array structure. A similar strategy has been employed in [40] where a force directed pairwise interchange algorithm is implemented on a two-dimensional array of processors.

A parallel floorplanning algorithm suitable for execution on a hypercube based distributed memory multiprocessor has been reported by Jayaraman and Rutenbar [78]. It is based on simulated annealing. Cohoon et al [41] have reported a parallel algorithm for floorplanning based on the theory of punctuated equilibria for genetic algorithms. The algorithm is suitable for execution on distributed memory multiprocessors. A parallel algorithm for floorplanning based on parallel depth first search strategies has been proposed by Arvindam et al [8].

Different hardware architectures for implementing Lee based routing algorithms have been described in [153] and [174]. Watanabe and Sugiyama [170] have given a parallel routing algorithm that can control path quality in two point connections. Rose [139], [140] has presented a parallel global routing algorithm based on enumerating a subset of all two-bend routes between two points. Zargham [177] has given a parallel algorithm based on a shared-memory multiprocessor for solving the problem of channel and switch-box routing. A parallel algorithm for global routing that is suitable for execution on a shared memory multiprocessor has been proposed by Nair et al on an array processor hardware [114].

Thulasiraman et al [163], [161] have proposed an approach for parallel layout compaction and wire balancing.

Bier and Pleszkun [20] presented a parallel algorithm for design rule checking. It works on the flattened representations of mask layouts and uses a data decomposition strategy. Gregoretti and Segall [67] have proposed a parallel bottom-up approach to design

rule checking which is suitable for execution on a shared memory multiprocessor. Carlson and Rutenbar [28], [29] have described massively parallel SIMD algorithms based on scanline approach for design rule checking on flattened representation. Blank et al [22] have proposed a bit mapped processor consisting of a three dimensional data structure of bits in hardware which can be programmed to perform DRC using the window checking scheme. Seiler [146] has presented a window processor which performs the DRC using rasterization hardware, a local checking hardware and an error reporting hardware. Carlson and Rutenbar [27] have proposed a scanline data structure processor for performing design rule checking.

A parallel algorithm for speeding up the task of circuit extraction was proposed by Levitin [98]. This approach involves splitting the design into equal size horizontal slices, and assigning one slice to each processor. Belkhale and Banerjee [15], [16], [17] have reported on the results of an implementation of a parallel algorithm for circuit extraction on an Intel iPSC/2 hypercube and have reported speedups of about 12 to 14 on 16 processors for various benchmark circuits. Tonkin [165] has presented a parallel version of a well-known sequential extractor.

Sadayappan and Viswanathan [143] have proposed a fine-grained parallel algorithm for circuit simulation on a vector multiprocessor, the Alliant FX/8 using a compiled/interpretive code approach. Trotter and Agarwal [166] have looked at the issues of implementing row-level parallel algorithms in distributed memory multiprocessors. Yuan et al [176] have implemented a PECSI circuit simulator on a hypercube using a distributed multifrontal method. Parallel algorithms for iterated timing analysis have been implemented in the MSPLICE program on the BBN Butterfly [48], [77]. A massively parallel algorithm for circuit simulation that is suitable for execution on an SIMD machine has been presented by Webber and Sangiovanni-Vincentelli [171].

Smart and Trick [149] have reported results of a parallel Waveform Relaxation

algorithm using the Gauss-Jacobi method of iterative solution on an Alliant FX/8 which is a 8 processor shared memory multiprocessor. A distributed memory parallel waveform relaxation algorithm based on a modified Gauss-Seidel algorithm has been proposed by Johnson and Zukowski [79]. White et al [172] have reported the implementation of the timepoint pipelining algorithm on a 10 processor Sequent Balance 8000 shared memory multiprocessor. Nakata et al [115] have developed a hardware accelerator for speeding up the direct method for circuit simulation.

The HSS program [14], developed by Barzilai et al at IBM, simulates a single good or faulty circuit on up to 32 patterns simultaneously. The FSS program developed by Tan [157] was designed to compute the expected signatures for circuits containing linear feedback shift registers. Bryant [26] has proposed two data parallel algorithms for the switch-level simulator COSMOS. Using a simple mode of concurrency, Smith et al [150] compared the effectiveness of several partitioning schemes for parallel logic simulation. Recently, several partitioning strategies have been proposed by Patil, Banerjee and Polychronopolous [130] and by Chamberlain and Franklin [33] which address the interprocessor communication cost explicitly during the partitioning.

Kravitz et al [85] have demonstrated the feasibility of mapping switch-level simulation onto a massively parallel SIMD Connection Machine. Soule and Blank [151] reported on an implementation of a simple parallel algorithm for event driven logic simulation on a shared memory multiprocessor. Mueller-Thuns et al [111] have reported on a parallel logic simulation algorithm for sequential circuits consisting of combinational gates and latches using a loosely synchronous model suitable for execution on distributed memory multiprocessors. Chandy and Misra [35] developed an alternative approach to parallel simulation which avoids the sending of null messages. Lubachevsky [100] proposed the use of a moving simulated time window to reduce the overhead associated with determining when it is safe to process an event. Soule and Gupta [152] have implemented

a parallel logic simulation algorithm using the conservative approach with deadlock detection and recovery. Arnold and Terman [7] used a greedy algorithm based on clustering for assigning the partitions to processors, followed by an iterative improvement stage which moves groups from one partition to another if there is a lot of communication across the groups in two partitions. More recently, Briner [25] has reported on parallel logic simulation algorithms on a shared memory multiprocessor which applied the Virtual Time/Time Warp paradigm more rigorously by using a combination of several optimizations.

IBM has built three generations of simulation engines that use circuit parallel algorithms using the compiled synchronous approach. The machines are the Logic Simulation Machine (LSM) [70], the Yorktown Simulation Engine [132] and the Engineering Verification Engine [52]. The MARS hardware accelerator [2], [3] uses the functional parallel approach to parallel logic simulation. Algorithms and implementations of logic simulation on vector processing machines are discussed in [112], [76], [136].

The simplest way to parallelize an automatic test pattern generation algorithm is to divide the fault list among the processors. Each processor then generates tests for each fault on its portion of the fault list until tests for all the faults have been generated. Such a scheme has been proposed by Chandra and Patel [34]. Communication among processors can be reduced or eliminated altogether by dividing the fault sets into independent fault sets [5]. Patil and Banerjee [128] have proposed a distributed memory parallel algorithm for test generation of combinational circuits which uses fault parallelism. Their method is a compromise between purely static and purely dynamic fault partitioning. Fujiwara and Inoue [57], [58] have presented an approach to parallel processing of test generation in a loosely coupled distributed network of workstations and analyzed the effects of allocation of target faults to processors, the optimal granularity (grain size of target faults), and the speedup ratio.

The PODEM-based test generator using parallel branch and bound has been imple-

mented on a 16 node Intel iPSC/2 hypercube [127]. Recently, Arvindam et al [10] have reported a parallel implementation of a similar parallel algorithm for ATPG on a network of Sun Workstations. Patil, Banerjee and Patel have reported on a parallel implementation of sequential circuit test generation using OR-parallel functional decomposition on a shared memory multiprocessor [129]. Motohara et al [110] implemented a version of AND-parallel functional decomposition on a distributed memory multicomputer system called LINKS-1 consisting of 50 processors connected as a tree of processors. Patil [126] has proposed an AND-parallel functional decomposition of test generation for sequential circuits suitable for shared memory multiprocessors.

A massively parallel SIMD algorithm for test generation has been recently proposed for the Connection Machine [104]. Duba et al [51] proposed a fault partitioned approach to parallel fault simulation based on a concurrent hierarchical fault simulator. Recently, Markas et al [103] have reported a distributed fault simulation algorithm on a heterogeneous network of workstations connected through a local area network. Ostapko et al [120] proposed a parallel algorithm for fault simulation using data parallel compiled simulation techniques. Recently, Kung and Lin [92] have proposed the Parallel Sequence Fault Simulation (PSS) algorithm for synchronous sequential circuits. Warshawsky and Rajski [169] have developed a parallel algorithm for fault simulation using input vector set partitioning that is suitable for use on distributed memory multiprocessor and networks of workstations.

One of the earliest works in parallel fault simulation based on circuit decomposition was proposed by Levendel, Menon and Patel [97]. Recently, Mueller-Thuns et al [111] have reported on a distributed memory parallel algorithm for fault simulation based on circuit decomposition for sequential circuits. A related parallel algorithm for fault simulation on a hypercube based distributed memory multiprocessor has been reported by Nelson [119]. More recently, Ghosh [62] has proposed a similar parallel circuit partitioned

algorithm called NODIFS for combinational and sequential circuits. Various techniques have been proposed in the past for parallel logic simulation using circuit partitioning [150], [151].

Narayan and Pitchumani [116], [117] have reported on massively parallel SIMD algorithms for fault simulation. Ozguner et al [121] have presented a parallel fault simulation algorithm using a pipelined decomposition. Chakradhar et al [32] have proposed a massively parallel algorithm for test generation using a neural network model of a logic circuit. Ishiura et al [75] proposed a dynamic two-dimensional parallel fault simulation technique where large vector lengths were obtained by utilizing both pattern and fault parallelism. Algorithms and implementations of fault simulation on vector processing machines are discussed in [168], [76], [75] and [122].

A parallel algorithm for two-level logic minimization based on ESPRESSO was proposed by Galivanche and Reddy [60]. Lim et al [99] have reported on a shared memory parallel implementation of a logic synthesis system using both circuit partitioned parallelism and intra-partition algorithmic partitioning. De, Ramkumar and Banerjee [46] proposed a novel asynchronous parallel algorithm for logic synthesis for distributed memory multiprocessors. A parallel algorithm for multilevel tautology checking based on a recursive divide and conquer sequential algorithm has been proposed by Hachtel and Moceyunas [69].

1.3 Integrated Layout Compaction and Wire Balancing

In this thesis, the integrated layout compaction and wire balancing (ILCWB) problem is used as an application to demonstrate the effectiveness of our parallel algorithms even though these algorithms can be used to solve many other problems that arise in engineering and industrial applications.

There are basically three approaches to layout compaction: compression ridges

(also known as shear-line), virtual grid and constraint graph compaction. For a discussion of these approaches, see [23], [37] and [96]]. They also contain an extensive list of references. We will use the constraint graph approach and formulate the problem in terms of the dual transshipment problem.

1.3.1 Constraint Graph Approach

The constraint graph approach is the most popular compaction method. This approach consists of two main steps: i) build the constraint graph to indicate the relative positions and the minimum distance required among the elements, ii) solve the constraint graph to minimize the chip area using the shortest/longest path method. Constraint graph compactors are generally one-dimensional compactors and require at least one X compaction pass and one Y compaction pass.

A directed graph is constructed to represent the layout where each node represents an element or a group of elements that move together and edges represent minimum spacing requirements between nodes. Since spacing constraints are translated into edge weights, each node must be at least longest path distance away from the boundary element to satisfy all the constraints. Hence a solution to the compaction problem is to keep each element at the longest path distance from the boundary assuming the boundary element is at zero coordinate. This solution also corrects design rule violations, if any, in the direction of compaction.

The main advantage of the constraint graph approach is the flexibility of the model. Incorporating user-defined constraints is very easy. The major drawback is that all objects are pushed toward the boundary as much as possible or evenly distributed. Features that lie on a longest path have no freedom to move, but features that do not lie on a longest path can be placed in a variety of possible locations. This additional freedom can be used to perform further optimizations on the layout. One popular optimization is total wire length minimization or wire balancing. Here, the features that do not lie on a longest

path are placed such that the total length of the horizontal (vertical) wires is minimized. Note that wires are not explicitly represented in the geometrical layout. Technological aspects can also enter this optimization.

The first graph based approach for layout compaction was reported by Cho et al [38]. But the details of their algorithms were not released to the public until much later. The CABBAGE [71] compactor of Hseueh is the most widely known constraint graph compactor and the general understanding of compaction was greatly enhanced by his comprehensive Ph.D. thesis. Since then a number of algorithms have been published using the constraint graph approach. Yoshimura [175] formulated the layout compaction and wire balancing problem as a dual transshipment problem and solved it using the simplex method. An extensive list of references are available in [24], [37], [96]. The constraint graph approach and the integrated layout compaction and wire balancing problems will be discussed further in Chapter 7.

1.4 Expressing Algorithms

To describe our parallel algorithms, we shall use the notation used by Akl [6] with slight modifications. This notation combines plain English with widely known programming constructs. For more information, see [6].

A parallel algorithm will normally consist of two kinds of operations: sequential and parallel. In describing the former, we use statements similar to those of a typical structured programming language. Examples of such statements include: **if** ... **then** ... **else**, **while** ... **do**, **for** ... **do**, assignment statements, input and output statements, and so on. The meanings of these statements are assumed to be known. A left-pointing arrow denotes the assignment operator; thus $a \leftarrow b$ means that the value of b is assigned to a. The logical operations **and**, **or**, **xor** (exclusive-or), and **not** are used in their familiar connotation. Thus, if a and b are two expressions, each taking one of the values **true** or **false**, then

1. (a **and** b) is **true** if *both a* and *b* are **true**; otherwise (a **and** b) is **false**;
2. (a **or** b) is **true** if *at least one* of *a* and *b* is **true**; otherwise (a **or** b) is **false**;
3. (a **xor** b) is **true** if *exactly one* of *a* and *b* is **true**; otherwise (a **xor** b) is **false**; and
4. (**not** a) is **true** if *a* is **false**; otherwise (**not** a) is **false**.

Parallel operations, on the other hand, are expressed by two kinds of statements:

1. When several steps are to be done at the same time, we write

   do steps *i* **to** *j* **in parallel**

   step i

   step i + 1

   .

   .

   .

   step j.

2. When several processors are to perform the same operations simultaneously until a boolean condition is satisfied, we write

   i) **while** (boolean expression) **do in parallel**

   {The operations to be performed by P_i are stated here}

   end while

   where boolean expression is evaluated first, or

   ii) **do in parallel**

   {The operations to be performed by P_i are stated here}

   while (boolean expression)

   where the statement(s) are executed once before the boolean expression is

evaluated.

3. When several processors are to perform the same operation simultaneously, we write

   i) **for** $i \leftarrow j$ **to** k **do in parallel**

      {The operations to be performed by P_i are stated here}

      end for

      where i takes every integer value from j to k, or

   ii) **for** $i \leftarrow r, s, ..., t$ **do in parallel**

      {The operations to be performed by P_i are stated here}

      end for

      where the integer values taken by i are enumerated, or

   iii) **for all** i **in** S **do in parallel**

      {The operations to be performed by P_i are stated here}

      end for

      where S is a given set of integers.

Comments in algorithm descriptions are surrounded with curly brackets (braces) { }, as shown in the preceding. Curly brackets are also used to denote a sequence of elements as, for example, in $A = \{a_0, a_1, ..., a_{n-1}\}$ or in $E = \{S_i \in S: s_i = m\}$. Both uses are fairly standard and easy to recognize from the context.

We also add an extra construct **lock** to indicate the semaphore write (or exclusive write) to a data structure. We assume that there are **lock** and **unlock** operations available to the processes either implicitly or explicitly so that only one process can lock and therefore write to that data structure while other processes will wait for this process to unlock the data structure when it finishes. This is expressed in two ways.

1. **lock** operation is assumed to be in effect for the duration of the execution of a statement. An **unlock** operation is implicitly assumed after the execution of the statement is over.

   lock statement

2. **lock** and **unlock** are explicitly stated. This is useful to execute a number of statements under locked condition.

   lock

   {The operations to be performed are stated here}

   unlock

1.5 Scope of the Thesis

This thesis is concerned with the development of efficient parallel algorithms for the transshipment problem and its dual, and their application in the integrated VLSI layout compaction and wire balancing problem. Three different parallel algorithms are developed and implemented on a shared-memory multiprocessor.

Chapters 2 - 7 of the thesis address the development of parallel algorithms for the transshipment problem and the dual transshipment problem. Chapter 2 discusses the transshipment problem in general and reviews the previous algorithms (sequential and parallel) in this area. It also outlines our three approaches. Chapter 3 discusses three basic network optimization algorithms that serve as building blocks in our parallel algorithms. A detailed discussion of their implementations is also given.

Our first parallel algorithm is for the transshipment problem. This is based on the primal-dual method. This parallel algorithm is discussed in Chapter 4.

Chapter 5 develops a new algorithm for the dual transshipment problem. This is based on the dual simplex method and is called the Modified Network Dual Simplex

(MNDS) method. This algorithm overcomes certain shortcomings of the dual simplex method which make it unattractive for parallelization.

Chapter 6 develops a novel approach, called the Cluster-Based Dual Simplex (CBDS) method, for the dual transshipment problem. This algorithm is based on a new characterisation of the optimum solutions of the dual transshipment problem.

Both MNDS and CBDS methods extensively use the notion of firings and are based on the theory of marked graphs.

In Chapter 7, we present results of an experimental evaluation of the three parallel algorithms - network primal dual, MNDS and CBDS methods - of Chapters 4 - 6. The algorithms have been implemented on the BBN Butterfly machine.

Chapter 8 of this thesis is concerned with the application of our parallel algorithms in VLSI layout compaction and wire balancing problems. In this chapter, the integrated layout compaction and wire balancing problem is defined. A unified formulation of this problem using the dual transshipment problem is developed. A comparative experimental evaluation of MNDS and CBDS algorithms as regards their performance in solving the integrated layout compaction and wire balancing problem is also given.

Finally, in Chapter 9, we summarize our contributions in this thesis and point out certain problems for future study.

CHAPTER 2

THE TRANSSHIPMENT PROBLEM AND ITS DUAL

Network optimization refers to the class of optimization problems defined on graphs or networks. These problems include the problem of constructing shortest paths, finding a maximum flow, constructing a minimum cost spanning tree, finding matchings in a network, etc. These problems occur in a variety of applications. While they are themselves significant in their fullness, they also occur as subproblems in several other applications. The transshipment problem (also known as the minimum cost flow problem) generalizes several of the network optimization problems [39].

The transshipment problem can be formulated as a linear program and therefore can be solved by the simplex method using any one of three approaches, namely, the primal, the dual or the primal-dual methods. The network simplex algorithm is an efficient implementation of the primal method for solving the transshipment problem and has been extensively studied in the literature. However, the dual method has received very little attention, since one can construct an optimum solution of the primal problem from an optimum solution of the dual. As we shall point out later that the network simplex algorithm is essentially sequential in nature and does not offer much scope for an efficient distributed implementation. This has motivated us to select the dual and the primal-dual methods as candidates to design efficient parallel algorithms for network optimization problems.

In this chapter, we give an introduction to the transshipment problem and its dual. We also give definitions of certain concepts that will be used in the remaining chapters.

2.1 The Transshipment Problem

The transshipment problem is defined as follows. We are given a network N interconnecting several nodes. Some of the nodes in N represent sources (e.g. manufacturing

centers) and are called *supply* nodes. Some of the others represent sinks and are called *demand* nodes. There may be several nodes which neither supply nor demand. These nodes are called *neutral* nodes. Each edge in the network is assigned a cost which represents the cost of transporting a unit quantity of a commodity. Given the supplies available at the sources, and the demands at the sinks, the *transshipment problem* is to arrive at a routing pattern for a given commodity so that the demands are satisfied at minimum cost.

In our formulation of this problem, we use the following notation.

The nodes of N will be denoted by the integers 1, 2, ..., n. Thus V = {1, 2, ..., n}. The supply or demand at a node i is denoted by w_i. Each edge $e = (i, j)$ is associated with a real number m_{ij} called the *token* of e. m_{ij} represents the cost of transporting unit quantity of the commodity along the edge e. Each edge (i, j) is also associated with a real number c_{ij} called *capacity* of the edge. c_{ij} represents the maximum quantity of the commodity that each (i, j) can accomodate. Thus, if x_{ij} represents the flow along the edge (i, j), then $0 \leq x_{ij} \leq c_{ij}$. It is assumed that the sum of the flows into a node is equal to the sum of the flows out of the node.

We follow the convention that w_i is negative if node i is a supply node; otherwise w_i is non-negative. Thus $-w_i$ will denote the total supply available at a supply node i. Clearly, $w_i = 0$ if node i is a neutral node.

With the different variables defined above we can formulate the transshipment problem as an LP problem as defined below.

Minimize : M X

subject to :

$$A^* X = W \tag{2.1}$$

$$0 \leq X \leq C \tag{2.2}$$

where

M = Row vector of edge tokens m_{ij}.

X = Column vector of edge flows x_{ij}.

A^* = −A, where A is the incidence matrix of the graph G.

C = Column vector of edge capacities c_{ij}.

W = Column vector of node supplies or demands.

Note that w_i = sum of the flows into node i - sum of the flows out of node i.

We assume without loss of generality that the total supply available is equal to the total demand. Thus,

$$\sum_i w_i = 0$$

As an example, a network N representing a transshipment problem is shown in Figure 2.1. The corresponding linear program is given below.

Minimize $3x_{12} + 5x_{13} + x_{15} + x_{23} + 4x_{42} + x_{43} + 6x_{53} + x_{54} + x_{56} + x_{62} + x_{64}$

subject to :

$$- x_{12} - x_{13} - x_{15} = -9$$

$$x_{12} - x_{23} + x_{42} + x_{62} = 4$$

$$x_{13} + x_{23} + x_{43} + x_{53} = 17$$

$$- x_{42} - x_{43} + x_{54} + x_{64} = 1$$

$$x_{15} - x_{53} - x_{54} - x_{56} = -5$$

$$x_{56} - x_{62} - x_{64} = -8$$

$$0 \le x_{12} \le 2$$

$$0 \le x_{13} \le 10$$

$$0 \le x_{15} \le 10$$

$$0 \le x_{23} \le 6$$

$$0 \le x_{42} \le 8$$

$$0 \le x_{43} \le 9$$

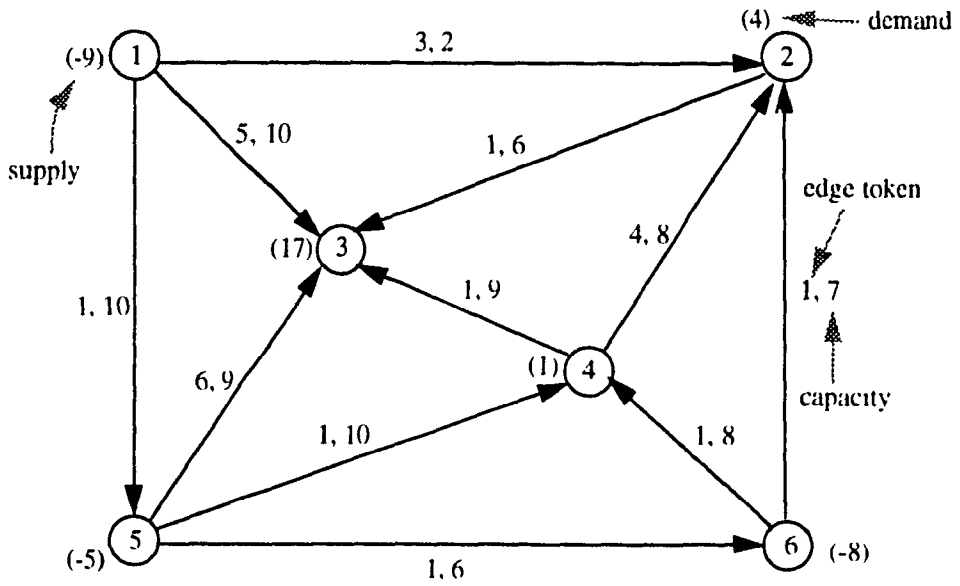
$$0 \le x_{53} \le 9$$

$$0 \le x_{54} \le 10$$

$$0 \le x_{56} \le 6$$

$$0 \le x_{62} \le 7$$

$$0 \le x_{64} \le 8$$



Node numbers are shown inside nodes.
Node Demands and edge tokens are as shown.

Figure 2.1 An Example: A Network N for the Transshipment Problem.

2.2 The Dual Transshipment Problem (DTP)

Associated with any linear programming problem there is a *dual problem.* The original problem is then called the *primal problem.* The dual of the transshipment problem has n dual variables $y_1, y_2, \ldots, y_n$. The optimum values for y_i's would maximize the sum $\sum_i w_i y_i$. Let A^t be the transpose of the incidence matrix A of G. If we assume that there is no upper bound on edge flows (that is, c_{ij}'s equal to ∞), then the dual transshipment problem is defined as follows.

Maximize: $W^t Y$

subject to

$$A^t Y \geq -M^t \tag{2.3}$$

$$Y \geq 0. \tag{2.4}$$

Note that the inequality in (2.3) corresponding to each (i, j) will be of the form

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

For instance, the inequality corresponding to edge (1, 2) in Figure 2.1 is

$$y_1 - y_2 \geq -3$$

2.3 Structure of Optimum Solutions for the Transshipment Problem and its Dual

An important result in linear programming theory is stated next.

If x_{ij}'s and y_i's represent optimum solutions for the primal and dual problems, respectively, then

$$x_{ij} = 0, \quad \text{if } y_i - y_j + m_{ij} > 0$$

$$x_{ij} = c_{ij}, \quad \text{if } y_i - y_j + m_{ij} < 0 \tag{2.5}$$

The above conditions are called the *complementary slackness* conditions. We can now say that for any optimum solution X for the transshipment problem the following are true:

iv) $A X = W$.

v) $0 \leq X \leq C$.

vi) There exists y_i's such that (2.5) is true.

At this point, it will be useful to explain the significance of the complementary slackness conditions.

Consider an edge (i, j) of N with flow x_{ij}. In linear programming theory, the quantity $y_i - y_j + m_{ij}$ is called the *relative cost coefficient* of (i, j). In this thesis, we shall refer to this as the *residual token* of (i, j). This quantity has an important role to play. If the flows in all the edges except (i, j) are kept unchanged and x_{ij} is changed to $x_{ij} + \Delta_{ij}$, then we can show that the objective function $\sum_{i,j} m_{ij}x_{ij}$ will change by $(y_i - y_j + m_{ij})\Delta_{ij}$. Since no further decrease in the value of the objective function of the transshipment problem is possible once an optimum is reached, it means that at that point

$$x_{ij} = 0, \text{ if } y_i - y_j + m_{ij} > 0$$

$$x_{ij} = c_{ij}, \text{ if } y_i - y_j + m_{ij} < 0. \qquad (2.6)$$

There are three distinct approaches to the transshipment problem - the primal, dual and primal - dual approaches [39]. The primal approach called the Network Simplex Method, starts with an X satisfying (2.1) and (2.2), repeatedly updates X (without violating (2.1) and (2.2)) until (2.5) is satisfied. The update of X is achieved through what is called a pivot operation. Note that each new X leads to a value for the objective function WX that is not greater than the previous value. As a sequential algorithm , the primal approach is known to be a very efficient one. However, unfortunately, it is not suited for a distributed or parallel implementation. The bottleneck here is the pivot operation which is inherently sequential in nature.

The Network Dual Simplex method is the same as the primal approach applied to the dual transshipment problem and hence it is also not suitable for a parallel implementation.

The primal - dual approach starts with an X and Y satisfying (2.2) and (2.5). It then updates X and Y (without violating (2.2) and (2.5)) until X satisfies (2.1). This approach is quite amenable to distributed and parallel implementations, since it uses the maximum flow and shortest path algorithms, as building blocks. Chapters 3 and 4 are concerned with the development of such a parallel algorithm.

Our algorithms in Chapters 5 and 6 deal with the dual transshipment problem. Both these algorithms extensively use the notion of node/cluster firings and results from the theory of marked graphs.

2.4 Basic Definitions

Given a directed graph G = (V, E), $(i, j) \in E$ will refer to the edge directed from i to j. 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. 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.

The notion of firing [42], [43], [162] will be used extensively in the development of our algorithms in this thesis. *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 become negative. Unless otherwise stated, firing would mean a positive 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 [164] may be referred to.

The variable Y in (2.3) and (2.4) is a column vector of node variables $y_1, y_2, \ldots, y_n$ and they will be referred to as *firing numbers*. Thus y_i will denote the firing number of node i. With these definitions, we can now see that the dual transshipment problem (DTP) seeks 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* of the DTP if Y satisfies constraint (2.3). 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 a firing no residual edge token should become negative, it follows

that a node i can be positively 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 firings of nodes i and j, then these firings can be done concurrently without making any resulting residual edge token negative. It is this property of firings 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 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 *dual pivot operation* is permissible if $W(S) \geq 0$ where $W(S) = \sum_{i \in S} W(i)$. If no dual pivot operation is permissible, then the solution Y is optimum. The dual pivot operation consists of firing the cluster S the maximum possible number of times and constructing a 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.

2.5 Summary

In this chapter, the transshipment problem and its dual are defined. The transshipment problem is also illustrated through an example. We have discussed the relationship between the optimum solutions for the transshipment problem and its dual. We have given some basic definitions which will be used in the development of our algorithms in the following chapters. For graph-theoretic definitions, Thulasiraman and Swamy [164] may be referred.

CHAPTER 3

BASIC NETWORK OPTIMIZATION ALGORITHMS

In this chapter, we discuss three basic network optimization algorithms and their parallel shared memory implementations. They are: Algorithm FEASIBLE to test the feasibility of the dual transshipment problem, Algorithm SHORTEST-PATH for the single-source shortest path problem, and the Maximum Flow (also known as Max Flow) Algorithm. These algorithms serve as building blocks for the algorithms to be developed in the following chapters. They also occur as subproblems in several other applications even though they are by themselves significant in their usefulness. Our discussion is based on [42], [43], [44], [63], [94].

3.1 Algorithm FEASIBLE

Consider the dual transshipment problem (DTP) defined in Section 2.2.

Maximize: $W^t Y$

subject to

$$A^t Y \geq -M^t \tag{3.1}$$

$$Y \geq 0 \tag{3.2}$$

Recall (Section 2.4) that a vector Y is called a feasible solution of the DTP, if $Y \geq 0$ and satisfies (3.1). If such a vector exists, then the given DTP is feasible. It can be shown [44] that a DTP is feasible if and only if the underlying graph G has no directed circuit of negative token. The following theorem also proved in [44] gives a solution to the DTP.

Theorem 3.1

Let $\sigma_i = \text{Max } \{0, -\min \{d_{ij}\}\}$, $i = 1, 2, \ldots, n$.

Then the vector $Y = (\sigma 1, ..., \sigma_n)$ is a feasible solution of the DTP if the graph G has no negative token directed circuit. ❑

Clearly, using a shortest path algorithm we can determine the vector Y defined in the above theorem. But such an algorithm would require constructing an auxiliary graph. In [44] an algorithm called Algorithm FEASIBLE which determines a feasible vector has been presented. This algorithm executes on the original graph itself and requires no auxiliary graph. An asynchronous distributed implementation of this algorithm as well as an efficient implementation using a synchronizer have also been presented in [44]. The time and message complexities of the synchronous version of this algorithm are $O(n)$ and $O(mn)$ respectively. Details of these algorithms and their proofs of correctness may be found in [44].

We now present an outline of a synchronous parallel version of Algorithm FEASIBLE. This algorithm computes σ_i's as defined in Theorem 3.1. Here, M will denote the initial token vector. The algorithm starts with $\sigma_i = 0$, for all i and proceeds in pulses. All processors execute the following sequence of instructions in parallel during a pulse. These instructions for node i are:

1. For each edge (i, j), compute the residual token $m_{ij} + \sigma_i - \sigma_j$.

   Set $r_{ij} = m_{ij} + \sigma_i - \sigma_j$.

2. Compute $Min\,(i) = Max\{0, \max_j \{-r_{ij}\}\}$

3. If Min $(i) > 0$, then fire node i, Min (i) times.

The above algorithm will terminate if during a pulse Min $(i) = 0$ for all i. It can be shown that the algorithm will terminate within n - 1 pulses, if the graph has no negative token directed circuit. If the algorithm does not terminate in n pulses, then the DTP is infeasible and in such a case we can also locate a negative token directed circuit by incorporating a suitable mechanism in the algorithm. A detailed discussion of these issues may

be found in [44].

Note that each processor i calculates the residual tokens in its local memory. This operation will require only reading the values of σ_i and σ_j and m_{ij} which can be done concurrently. So to implement this step, no lock operations will be required. It was found that the performance of the algorithm improves considerably because of this.

Also, we need no explicit synchronizer mechanism unlike in the distributed implementation discussed in [44]. The inherent synchronization available when one spawns different processes in the BBN parallel computer itself serves as a synchronizer.

The following data structures are used in the parallel implementation of Algorithm FEASIBLE.

token	:	a 2-dimensional array of integers. *token [i] [j]* represents the value of token of edge (i, j) directed from node i to node j if this edge exists in the graph G; otherwise it contains a special value INFINITY outside the range of edge tokens.
outNodes	:	an array of sets. *outNodes [i]* is a set containing all the out-nodes at node i.
firing_no	:	an array of integers. At any time, *firing_no [i]* represents the current firing number of node i.

Algorithm FEASIBLE is formally presented as follows.

Algorithm FEASIBLE

procedure initialize_feasible (*i*)

{ This procedure initializes *firing_no [i]* to the value of the most negative token of an outgoing edge at node *i*. If there is no such edge at node *i*, then *firing_no [i]* to zero. }

fno ← 0;

for all *j* **in** *outNodes [i]* **do**

if (*token [i] [j]* < *fno*)

then *fno* ← *token [i] [j]*

end if;

end for;

firing_no [i] ← - *fno;*

if (*fno* < 0)

then *notFinished* ← TRUE

end if;

end procedure { initialize_feasible }

procedure get_most_negative_edge_token (*i*)

{ This procedure calculates the new relative tokens for all the out-edges at node *i* and then returns the the value of the most negative edge token }

fno ← 0;

for all *j* **in** *outNodes [i]* **do**

temp ← *token [i] [j]* + *firing_no [i]* - *firing_no [j];*

if (*temp* < *fno*)

```
        then fno ← temp
        end if;
    end for;
    return fno;
end procedure { get_most_negative_edge_token }
procedure fire_if_negative_edge (i)
{ It fires node i if there is at least one negative token at an outgoing edge }
    new_fno ← get_most_negative_edge_weight (i);
    if (new_fno ≠ 0)
    then
        notFinished ← TRUE;
        firing_no [i] ← firing_no [i] - new_fno;
    end if;
end procedure { fire_if_negative_edge }
procedure feasible ()
{ main procedure }
    notFinished ← FALSE;
    for i ← 0 to N - 1 do in parallel
        initialize_feasible (i)
    end for;
    npulses ← 0;
    while (notFinished and npulses < N) do
        notFinished ← FALSE;
```

```
        npulses ← npulses + 1;
        for i ← 0 to N - 1 do in parallel
            fire_if_negative_edge (i)
        end for;
    end while;
    if (npulses < N)
    then return TRUE
    else return FALSE
    end if;
end procedure { feasible }
```

3.2 Algorithm SHORTEST-PATH

Consider a connected directed graph G with tokens assigned to the edges of G. Recall that (Section 2.4) the token of a directed path in G denotes the sum of the tokens of the edges in the path. A minimum token directed *s-t* path in G is called a shortest path from *s* to *t*. The token of a shortest directed *s-t* path, called the distance from *s* to *t*, is denoted as *d(s, t)*. In the *single-source shortest path problem*, one is interested in finding the shortest paths from a specified node *s* to all the nodes in G.

The Bellman-Ford-Moore (BFM) algorithm [164] for the single-source shortest path problem is very elegant and easy to present. It is also amenable for parallel implementation. Initially this algorithm assigns a label DISTANCE (s) = 0 to the node s and assigns DISTANCE (i) = ∞, for every other node i. The algorithm then repeatedly performs the following operation:

DISTANCE -UPDATE:

Select an edge $e = (i, j)$ such that DISTANCE (j) > DISTANCE $(i) + m_{ij}$.

Set DISTANCE (j) = DISTANCE $(i) + m_{ij}$.

The algorithm terminates when DISTANCE -UPDATE is no longer applicable. Termination occurs iff there are no negative token directed circuits in G. At termination DISTANCE (i) gives the token of a shortest path from s to i.

An efficient implementation of the BFM algorithm is as follows. Order the nodes as 1, 2, ..., n. Pick nodes in this order, and for each node i selected, examine all the edges directed out of i and perform DISTANCE -UPDATE on these edges whenever it is applicable. After one such sweep (examination) of all the nodes, perform additional sweeps until an entire sweep produces no changes in the node labels.

A variant of the BFM algorithm that is amenable for a distributed implementation is presented in [94]. This can also be easily extended to the multiple source multiple destination case.

In the following, we present a parallel shared memory implementation of the above algorithm. Since the parallel algorithms developed in the following chapters would require shortest path computations only after an application of Algorithm FEASIBLE, the graphs under consideration at that point will have no negative tokens. Therefore, termination detection in our shortest path algorithm is much simpler than the one presented in [94]. The algorithm will be terminated when DISTANCE -UPDATE is no longer applicable. Also, successful termination of this shortest path algorithm will occur in less than n pulses.

The following data structures are used.

token : a 2-dimensional array of integers. *token [i] [j]* represents the value of token of edge (i, j) directed from node i to node j if this edge exists in the graph G; otherwise it contains a special value INFINITY outside the range of edge tokens.

outNodes : an array of sets. *outNodes [i]* is a set containing all the outnodes at node i.

pred : an array of integers. At termination, *pred [i]* indicates the predecessor of node i in a shortest path from the source to node i. Each *pred [i]* is initialized to i.

distance : an array of integers. At termination, *distance [i]* indicates the shortest distance of node i from the source. Each *distance [i]* is initialized to INFINITY.

newDistance : an array of integers. *newDistance [i]* indicates the change in the shortest distance for node i since the last update. *newDistance [i]* is initialized to zero if node i is a source; otherwise it is initialized to INFINITY.

There are two pulses in this algorithm. First one is the initialization pulse and the second pulse is the main iterative one. The second pulse is repeated until all nodes reach their shortest distances. This can be detected when *distance [i]* = *newdistance [i]* for all i. This will happen within n pulses since the graph contains no negative token directed circuits.

In the first (initialization) pulse, each node i will initialize its *distance [i]* to INFINITY. The source node will initialize its *newDistance* variable to zero while all other nodes will initialize their *newDistance* variables to INFINITY.

In each iteration of the second pulse, every node *i* inspects if there is a change in its distance since the last update and if so, then node *i* will calculate the new shortest distance for each of its outnodes and propagate this new distance to each outnode if it is less than the previous distance value at that node.

Following is a formal presentation of our parallel shortest path algorithm.

Algorithm SHORTEST-PATH

```
procedure sp_initialize (i)
    distance [i] ← INFINITY;
    pred [i] ← i;
    if (i ≠ SOURCE)
    then newDistance [j] ← INFINITY
    else
        newDistance [j] ← 0;
        notFinished ← TRUE; { This condition is used in Chapter 5 }
    end if;
end procedure { sp_initialize }
procedure propagate_shortest_distances (i)
{ Do DISTANCE _UPDATE operation for node i }
    d ← newDistance [i];
    if (d < distance [i])
    then
        distance [i] ← d;
        notFinished ← TRUE;
        for all j in outNodes [i] do
```

```
            t ← token [i] [j] + d;
            lock j;
            if (t < newDistance [j])
            then
                newDistance [j] ← t;
                pred [j] ← i;
            end if;
            unlock j;
        end for;
    end if;
end procedure { propagate_shortest_distances }
procedure shortest_paths ()
{ main algorithm }
    for i ← 0 to N - 1 do in parallel
        sp_initialize (i)
    end for;
    do
        notFinished ← FALSE;
        for i ← 0 to N - 1 do in parallel
            propagate_shortest_distances (i)
        end for;
    while (notFinished);
end procedure { shortest_paths }
```

3.3 Algorithm MAX-FLOW

A transport network represents a model for transportation of a commodity from its production center to its market through communication routes. The network is thus a connected directed graph N=(V, E) with no self loops (cannot transport to itself). N has to satisfy the following conditions.

There is only one node with zero indegree; this is designated as the source (production center) and is denoted as s. There is only one node with zero outdegree; this is designated as the sink (market) and is denoted as t. Every directed edge $e = (i, j)$ in N is assigned a non-negative real number c_{ij}, the capacity of (i, j). $c_{ij} = 0$ if there is no edge directed from i to j. Every directed edge $e=(i, j)$ in N is also assigned a non-negative real number called the flow f_{ij} on (i, j).

A flow f through a transport network N is an assignment of non-negative real numbers f_{ij} to the edges (i, j) such that the following conditions are satisfied:

1. *capacity constraint:* $0 \leq f_{ij} \leq c_{ij}$, for $(i, j) \in$ E.
2. *conservation constraint:* For each node i, except the source s and the sink t, the material transported into i is equal to the material transported out of i.

The value val(f) of a flow f is defined as

$$val(f) = \sum_i f(s, i) = \sum_i f(i, t)$$

A flow f^* in a transport network N is said to be maximum if there is no flow f in N such that val(f) > val(f^*). The *maximum flow (in short, the max-flow) problem* is to find a maximum flow in a transport network.

The Push-Relabel Preflow Algorithm: Goldberg and Tarjan

The earliest algorithm for the max flow problem was due to Ford and Fulkerson [56]. Several variations of the Ford-Fulkerson algorithm were subsequently presented resulting in algorithms with better complexities. Recently Goldberg and Tarjan [63], [64] presented an approach that is fundamentally different from that of Ford and Fulkerson. This algorithm is quite amenable for distributed/parallel implementation. We now proceed to present Goldberg and Tarjan's max-flow algorithm.

Let N=(V, E) be a network with each edge assigned a non-negative real capacity. Without loss of generality assume that N has no multiple edges. If there is an edge from a node i to a node j, this edge is unique by the assumption and is denoted by (i, j). A *pseudoflow* is a function $f: E \rightarrow R$ that satisfies the following constraints:

$$f_{ij} \leq c_{ij}, \qquad \forall (i, j) \in E \text{ (capacity constraint)}$$

$$f_{ji} = -f_{ij}, \qquad \forall (i, j) \in E \text{ (antisymmetry constraint)}$$

We let $c_{ji} = 0$ if $(i, j) \in E$. Given a pseudoflow f, the excess function $e_f: V \rightarrow R$ is defined by,

$$e_f(i) = \sum_{k \in V} f_{ki}$$

Thus $e_f(i)$ is the net flow into i. A node i has *excess* if $e_f(i)$ is positive. This indicates that some amount of flow can be pushed out from node i. A node i has *deficit* if $e_f(i)$ is negative.

Given a pseudoflow f, the *residual capacity* function $c_f: E \rightarrow R$ is defined by $c_f(i, j) = c_{ij} - f_{ij}$. The *residual graph* with respect to a pseudoflow f is given by $G_f = (V, E_f)$, where $E_f = \{(i, j) \in E, \mid c_f(i, j) > 0\}$. Edge (i, j) is a *residual edge* if $c_f(i, j) > 0$.

A *preflow* f is a pseudoflow f such that the $e_f(i) \geq 0$ for all nodes i other than s and t.

The push-relabel preflow algorithm of Goldberg and Tarjan starts with a preflow and a distance labeling, and uses two operations, *pushing* and *relabeling*, to update the preflow and the labeling, repeating them until a maximum flow is found. For a given preflow f, a valid distance labeling is a function d from the nodes to the non-negative integers such that $d(s) = n$, $d(t) = 0$ and $d(i) \leq d(j) + 1$ for all residual edges (i, j).

A node i is said to be *active* if $i \notin \{s, t\}$ and $e_f(i) > 0$. An edge (i, j) is *admissible* if $c_f(i, j) > 0$ and $d(i) = d(j) + 1$.

The push-relabel algorithm begins with an initialization phase. The flow on each edge leaving the source is set equal to the edge capacity, and all other edges not incident on the source have zero flow. For each node j, the excess $e_f(j)$ is calculated. It is clear that since some flow is pushed from the source, there exists at least one node with positive excess. So there exists at least one active node. Each node $j \in V - \{s\}$ is assigned an initial labeling $d(j) = 0$. For node s, $d(s) = n$. Then an update operation is selected and applied to an active node. This process continues until there are no more active nodes at which point the algorithm terminates, with a preflow f with no active nodes. f is a maximum flow at termination.

We next present the update operations. The push operation modifies the preflow f and the relabel operation modifies the valid distance labeling d.

Push (i, j).

Applicability

i is active, $c_f(i, j) > 0$ and $d(i) = d(j) + 1$.

Action

send $\delta = \min(e_f(i), c_f(i, j))$ units of flow from i to j;

$f_{ij} \leftarrow f_{ij} + \delta; f_{ji} \leftarrow f_{ji} - \delta;$

$e_f(i) \leftarrow e_f(i) - \delta; e_f(j) \leftarrow e_f(j) + \delta;$

Relabel (*i*).

Applicability

i is active **and** $\forall j \in V, c_f(i, j) > 0 \Rightarrow d(i) \leq d(j)$.

Action

$$d(i) \leftarrow \min_{c_f(i,j) > 0} \{d(j) + 1\}$$

(If this minimum is over an empty set, $d(i) \leftarrow \infty$).

An efficient implementation of the push/relabel algorithm is discussed next. In this implementation, an unordered pair $\{i, j\}$ such that $(i, j) \in E$ or $(j, i) \in E$ is an *undirected edge* of G. Each undirected edge $\{i, j\}$ is associated with three values c_{ij}, c_{ji} and f_{ij} $(= -f_{ji})$. Each node *i* has a list of the incident edges $\{i, j\}$, in fixed but arbitrary order. Thus each edge $\{i, j\}$ appears in exactly two lists, the one for *i* and the one for *j*. Each node *i* has a *current edge* $\{i, j\}$ which is the current candidate for a pushing operation from *i*. The max-flow algorithm repeats the *push/relabel* operation given below until there are no more active nodes. The push/relabel operation combines the basic push and relabel operations.

Push/Relabel (*i*).

Applicability

i is active.

Action

Let $\{i, j\}$ be the current edge of i.

If *push* (i, j) is applicable **then** *push* (i, j)

else

If $\{i, j\}$ is not the last edge on the edge list of i

then replace $\{i, j\}$ as the current edge of i by the next edge on the edge list of i;

else begin

make the first edge on the edge list of i the current edge;

relabel (i);

end.

In a parallel implementation of the push-relabel preflow max-flow algorithm, the nodes perform in parallel the following pulses (sets of sequential instructions) in that order.

1. Push
2. Relabel, if necessary
3. Update flows and compute excess flows

These pulses are repeated until termination. It should be ensured that relabeling is done only after all the nodes have completed the push operations. For this reason, the push/relabel (i) operation as described above is not appropriate for use in a parallel environment. Furthermore, on the completion of push operations by the nodes, each node should have information about the flows pushed into itself from the adjacent nodes. This is essential to update the edge flows and to compute the excess flow at each node. This means that during a push pulse, each node should consider all incident edges for a possible push operation and inform the adjacent nodes about the flows pushed. This also means that

if even when a push operation is not applicable along an edge (i, j), node i should inform so node j. This is done by pushing a flow of zero value along (i, j). These issues are taken into consideration in the presentation of the parallel max flow algorithm described next.

As regards termination, the algorithm may be terminated when there are no active nodes at the end of a push pulse. An alternate termination detection scheme based on the following theorem is more elegant. It is this scheme we have used in our algorithm development.

Theorem 3.2

If at any pulse, the total flow out of the source is equal to the total flow into the sink, then at that pulse and all subsequent pulses there will be no active nodes. ❑

We next give a formal presentation of a parallel shared memory implementation of the push/relabel max-flow algorithm. The following data structures are used to describe the algorithm.

totalOutFlow	:	an integer variable to indicate the total flow out of the source. It is initialized to zero.
totalInFlow	:	an integer variable to indicate the total flow into the sink. It is initialized to zero. When *totalInFlow* equals *totalOutFlow*, the algorithm is terminated.
isRelabelNeeded	:	an array of boolean variables. *isRelabelNeeded [i]* equal to TRUE indicates that the node i needs to be relabeled.
outNodes	:	an array of sets. *outNodes [i]* is a set containing all the out-nodes at node i.

inNodes : an array of sets. *inNodes [i]* is a set containing all the innodes at node *i*.

label : an array of integers. *label [i]* indicates the label of node *i*. The *label* of source is initialized to *n* while the *labels* for all other nodes are initialized to zero.

tempLabel : an array of integers. *tempLabel [i]* is the label of node *i* after a relabeling operation.

excessFlow : an array of integers. *excessFlow [i]* indicates the excess flow at node *i* at any time. Each *excessFlow [i]* is initialized to zero.

capacity : a 2-dimensional array of integers. *capacity [i] [j]* indicates the capacity of the edge *(i, j)* if *(i, j)* is in G; otherwise it is equal to zero.

outFlow : a 2-dimensional array of integers. *outFlow [i] [j]* indicates the flow from node *i* to node *j* along the edge *(i, j)*, if *(i, j)* is an outgoing edge at *i*. Each *outFlow [i] [j]* is initialized to zero.

inFlow : a 2-dimensional array of integers. *inFlow [i] [j]* indicates the flow from node *i* to node *j* if *(j, i)* is an incoming edge at node *i*. *inFlow [i] [i]* is equal to *–outFlow [j] [i]*. Each *inFlow [i] [j]* is initialized to zero.

outDelta : a 2-dimensional array of integers. *outDelta [i] [j]* indicates the flow pushed from node *i* to node *j* along outgoing edge *(i, j)*. Each *outDelta [i] [j]* is initialized to zero.

inDelta : a 2-dimensional array of integers. *inDelta [i] [j]* indicates the negative value of the flow pushed from node *i* to node *j* along incoming edge (*j*, *i*). Each *inDelta [i] [j]* is initialized to zero.

The parallel max-flow algorithm is formally presented next.

Algorithm MAX-FLOW

```
procedure mf_initialization ( i )
    label [i] ← 0;
    tempLabel [i] ← 0;
    excessFlow [i] ← 0;
    isRelabelNeeded [i] ← FALSE;
    for j ← 0 to n - 1 do
        outFlow [i] [j] ← 0;
        inFlow [i] [j] ← 0;
        outDelta [i] [j] ← 0;
        inDelta [i] [j] ← 0;
    end for;
end procedure { mf_initialization }
procedure start_with_source ()
{ The source node will start the process by sending the initial flow }
    totalOutFlow ← 0;
    totalInFlow ← 0;
    s ← SOURCE;
    label [s] ← n;
```

```
    for all j in outNodes [s] do
        outFlow [s] [j] ← capacity [s] [j];
        inDelta [j] [s] ← – capacity [s] [j];
        excessFlow [j] ← excessFlow [j] – inDelta [j] [s];
        inFlow [j] [s] ← inFlow [j] [s] + inDelta [j] [s];
        totalOutFlow ← totalOutFlow + capacity [s] [j];
    end for;
end procedure { start_with_source }
procedure isLastEdge (i, ce)
{ is the current edge 'ce' the last edge at node i }
    if (inNodes [i] ≠ ∅)
    then
        if (ce is the last member of inNodes [i])
        then return TRUE
        end if
    elsif (ce is the last member of outNodes [i])
    then return TRUE
    end if;
    return FALSE;
end procedure { isLastEdge }
procedure determine_next_edge (i, ce)
{ determine the next edge after the current edge 'ce' at node i }
    if (ce ∈ outNodes [i])
```

```
then
    if (ce is not the last member of outNodes [i])
    then temp ← next member in outNodes [i] after ce
    elsif (inNodes [i] ≠ ∅)
    then temp ← first member of inNodes [i]
    else
        temp ← first member of outNodes [i];
        if (excessFlow [i] > 0)
        then isRelabelNeeded [i] ← TRUE
        end if;
    end if
else
    if (ce is not the last member of inNodes [i])
    then temp ← next member in inNodes [i] after ce
    elsif (outNodes [i] ≠ ∅)
    then temp ← first member of outNodes [i]
    else
        temp ← first member of inNodes [i];
        if (excessFlow [i] > 0)
        then isRelabelNeeded [i] ← TRUE
        end if;
    end if
end if;
```

```
    return temp;
end procedure { determine_next_edge }
procedure push ( i )
    lastEdge ← FALSE;
{ Make the first available edge at node i as the current edge 'ce' }
    if (outNodes [i] ≠ ∅)
    then ce ← first member in outNodes [i]
    else ce ← first member in inNodes [i]
    end if;
    if (excessFlow [i] = 0)
    then
        for all j in outNodes [i] do
            inDelta [j] [i] ← 0
        end for;
        for all k in inNodes [i] do
            outDelta [k] [i] ← 0
        end for;
    end if;
    while (excessFlow [i] ≠ 0 and not lastEdge) do
        if (ce ∈ outNodes [i])
        then residualcap ← capacity [i] [ce] - outFlow [i] [ce]
        else residualcap ← - inFlow [i] [ce]
        end if;
```

```
        if (residualcap > 0 and label [i] = label [ce] + 1)
        then
            if (excessFlow [i] < residualcap)
            then temp ← excessFlow [i]
            else temp ← residualcap
            end if;
            excessFlow [i] ← excessFlow [i] – temp;
            if (ce ∈ outNodes [i])
            then
                inDelta [ce] [i] ← – temp;
                outFlow [i] [ce] ← outFlow [i] [ce] + temp;
            else
                outDelta [ce] [i] ← – temp;
                inFlow [i] [ce] ← inFlow [i] [ce] + temp;
            end if;
        end if;
        if (isLastEdge (i, ce))
        then lastEdge ← TRUE
        else ce ← determine_next_edge (i, ce) {Make next edge as current edge}
        end if;
    end while;
end procedure { push }
```

```
procedure relabel ( i )
    if (isRelabelNeeded [i])
    then
        dist ← INFINITY;
        for all j in outNodes [i] do
            residualcap ← capacity [i] [j] – outFlow [i] [j];
            if (residualcap > 0 and label [j] < dist)
            then dist ← label [j]
            end if;
        end for;
        for all k in inNodes [i] do
            residualcap ← – inFlow [i] [j];
            if (residualcap > 0 and label [k] < dist)
            then dist ← label [k]
            end if;
        end for;
        if (dist < INFINITY)
        then tempLabel [i] ← dist + 1
        end if;
    end if
end procedure { relabel }
procedure broadcast_label ( i )
    if (isRelabelNeeded [i])
```

```
        then label [i] ← tempLabel [i];
        end if
    end procedure { broadcast_label }
    procedure calculate_total_out_flow ()
    { Calculates the total flow out of the source }
        totalOutFlow ← 0;
        s ← SOURCE;
        for all j in outNodes [s] do
            outFlow [s] [j] ← outFlow [s] [j] + outDelta [s] [j];
            totalOutFlow ← totalOutFlow + outFlow [s] [j];
        end for;
    end procedure { calculate_total_out_flow }
    procedure calculate_total_in_flow ()
    { Calculates the total flow into the sink }
        totalInFlow ← 0;
        t ← SINK;
        for all k in inNodes [t] do
            inFlow [t] [k] ← inFlow [t] [k] + inDelta [t] [k];
            totalInFlow ← totalInFlow - inFlow [t] [k];
        end for;
    end procedure { calculate_total_in_flow }
    procedure calculate_excess_flow ( i )
        for all j in outNodes [i] do
```

```
        excessFlow [i] ← excessFlow [i] – outDelta [i] [j];
        outFlow [i] [j] ← outFlow [i] [j] + outDelta [i] [j];
    end for;
    for all k in inNodes [i] do
        excessFlow [i] ← excessFlow [i] – inDelta [i] [k];
        inFlow [i] [k] ← inFlow [i] [k] + inDelta [i] [k];
    end for;
end procedure { calculate_excess_flow }
procedure update_flows ( i )
    if (i = SOURCE)
    then calculate_total_out_flow ()
    elsif (i = SINK)
    then calculate_total_in_flow ()
    else calculate_excess_flow ( i )
    end if
end procedure { update_flows }
procedure max_flow ()
{ main procedure }
    for i ← 0 to n - 1 do in parallel
        mf_initialization ( i )
    end for;
    start_with_source ();
    while (totalOutFlow ≠ totalInFlow) do
```

```
        for i ← 0 to n - 1 do in parallel
            if (i ≠ SOURCE and i ≠ SINK)
            then push ( i )
            end if
        end for;
        for i ← 0 to n - 1 do in parallel
            if (i ≠ SOURCE and i ≠ SINK)
            then relabel ( i )
            end if
        end for;
        for i ← 0 to n - 1 do in parallel
            if (i ≠ SOURCE and i ≠ SINK)
            then broadcast_label ( i )
            end if
        end for;
        for i ← 0 to n - 1 do in parallel
            update_flows ( i )
        end for;
    end while;
end procedure { max_flow }
```

3.4 Summary

In this chapter, we have discussed three basic network optimization algorithms and their parallel implementations. These algorithms will serve as building blocks in the parallel algorithms to be developed in the following chapters. Algorithm FEASIBLE to test

feasibility of the DTP will be used in the two parallel algorithms for the DTP discussed in Chapters 5 and 6. As will be shown in Chapter 4, this algorithm can also be adapted to initialize the primal-dual method and is quite an attractive alternative to the traditional approach to primal-dual initialization. The shortest path algorithm will be required for both the primal-dual method as well as for the modified network dual simplex method developed in Chapter 5. The max-flow algorithm will be required in the development of the primal-dual method. Besides this application, the max-flow algorithm has several applications in network optimization since many network optimization problems can be transformed into equivalent max-flow problems. Some of these may be found in [164]. Though Goldberg and Tarjan's max-flow algorithm is very elegant and amenable for a parallel implementation, its parallelization does not permit incorporating certain features of the elegant sequential implementation analyzed in [64].

CHAPTER 4

PARALLEL NETWORK PRIMAL DUAL METHOD

In this chapter, we discuss the network primal-dual method for the transshipment problem and its implementation on the shared memory model. This work has been reported in [161]. First, we briefly recall the definition and formulation of the transshipment problem (presented in Chapter 2) and the basic structure of an optimum solution for the problem.

The transshipment problem is concerned with a network N having the underlying directed graph G = (V, E). The network N has n nodes. Some of these nodes are supply nodes, some are demand nodes and the others are neutral nodes. Each node i is associated with a real number w_i representing the demand or supply of a commodity at node i. We follow the convention that w_i is negative if node i is a supply node; otherwise w_i is non-negative. It is assumed that the total supply is equal to the total demand. Each edge $e = (i, j)$ is associated with a real number c_{ij} called the capacity of (i, j) and another real number m_{ij} called the token of e. The capacity c_{ij} represents the maximum quantity of the commodity that each (i, j) can accomodate and m_{ij} represents the cost of transporting unit quantity of the commodity along the edge e. Thus, if x_{ij} represents the flow along the edge (i, j), then $0 \leq x_{ij} \leq c_{ij}$. It is assumed to satisfy the conservative constraint: The sum of the flows into a node is equal to the sum of the flows out of the node. The transshipment problem is to arrive at a pattern of tranporting the commodity from the supply nodes to the demand nodes at minimum cost. The problem can be formulated as the following linear programming problem, where M, X, A^* and W are as defined in Section 2.1.

Minimize : M X

subject to :

$$A^* X = W \tag{4.1}$$

$$0 \leq X \leq C \tag{4.2}$$

Associated with a linear program, there is a dual problem. The original problem is called the primal problem. The dual problem has n dual variables, $y_1, y_2, \ldots, y_n$, one for each one of the n nodes. The optimum values for y_i's would maximize $\Sigma w_i y_i$. We shall refer to $y_1, y_2, \ldots, y_n$ as firing numbers since they indeed play the role of firing numbers as will be seen in Chapters 5 and 6.

If x_{ij}'s and y_i's represent the optimum solutions for the primal and dual problems respectively, then

$$\begin{aligned} x_{ij} &= 0\,, \qquad \text{if } y_i - y_j + m_{ij} > 0 \\ x_{ij} &= c_{ij}\,, \qquad \text{if } y_i - y_j + m_{ij} < 0 \end{aligned} \tag{4.3}$$

Thus, for any optimum solution X for the transshipment problem, the following are true:

i) $A X = W$

ii) $0 \leq X \leq C$

iii) There exists y_i's such that (4.3) is true.

The primal-dual approach for solving the transshipment problem starts with an X and Y satisfying (4.2) and (4.3). It then repeatedly updates X and Y (without violating (4.2) and (4.3)) until X satisfies (4.1). This approach is quite amenable to distributed and parallel implementations, since it uses the maximum flow and shortest path algorithms, as building blocks.

4.1 Primal-Dual Method

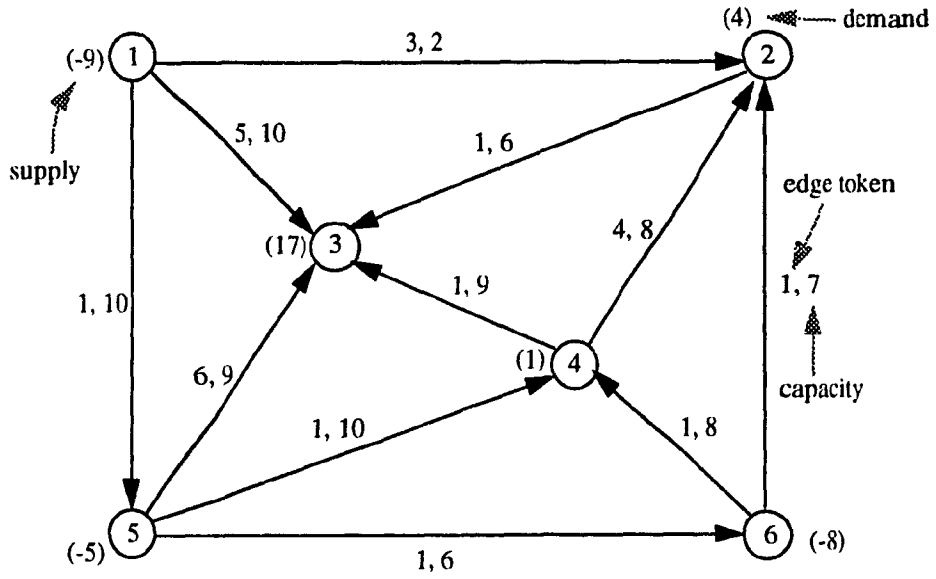
The primal-dual method for the transshipment problem consists of three main steps:

1. Initialization.

2. Updating *Y*.

3. Updating *X*.

Each of these steps is explained in detail in the following sections. The graph G in Figure 4.1 is used to illustrate the different steps in the Primal-Dual method. Note that this is the same as the graph used in Chapter 2 to illustrate the transshipment problem (See Figure 2.1).



Node numbers are shown inside nodes.
Node Demands and edge tokens are as shown.

Figure 4.1 An Example: A Network N for the Transshipment Problem.

4.1.1 Initialization of the Primal-Dual Method

In this step, a pair of vectors *X* and *Y* is selected such that the complementary slackness conditions are satisfied.

i) $0 \le x_{ij} \le c_{ij}$

ii) $x_{ij} = \begin{cases} 0, \text{ whenever } y_i - y_j + m_{ij} \ge 0 \\ c_{ij}, \text{ whenever } y_i - y_j + m_{ij} < 0 \end{cases}$

The traditional approach to this initialization problem discussed in [39] is to apply the primal dual method to a new network constructed from the given network. We now point out that Algorithm FEASIBLE discussed in Section 3.1 can be adapted to solve this problem.

Three cases arise.

Case 1 : All m_{ij}'s are positive.

In this case, selecting all x_{ij}'s and y_i's equal to zero will result in a pair of X and Y satisfying the complementary slackness conditions.

Case 2 : $c_{ij} = \infty$ for all edges *(i, j)*.

In this case, we need to find y_i's such that $y_i - y_j + m_{ij} \ge 0$ for all edges *(i, j)*. These y_i's can be found by applying Algorithm FEASIBLE on G. The corresponding X vector is to be selected with all x_{ij}'s equal to zero.

Case 3 : c_{ij} finite for some edges *(i, j)*.

In this case, we need to find y_i's such that for all edges *(i, j)* with $c_{ij} = \infty$, $y_i - y_j + m_{ij} \ge 0$. Such y_i's can be obtained by applying Algorithm FEASIBLE on G after removing all those edges *(i, j)* with c_{ij} finite (see Case 2). The X vector can then be selected satisfying the complementary slackness conditions.

Note that in all these cases, we ensure that $y_i - y_j + m_{ij} \ge 0$ whenever $c_{ij} = \infty$. Furthermore, we can apply Algorithm Feasible without explicitly removing certain edges as required in Case 3.

4.1.2 Updating the Dual Vector Y

Given a pair of X and Y vectors satisfying (4.2) and (4.3), we now explain how we can update X and Y without violating these conditions.

From X and W we first calculate the new demand vector W´as follows.

$$w'_i = w_i \text{ - net flow into node } i$$

$$= w_i - \sum_j a_{ji} x_{ji} + \sum_j a_{ij} x_{ij}$$

Each w'_i then gives the current supply available or the current demand at node i. Following Chvatal [39], we shall call a node i wet, balanced or dry if w'_i is negative, zero or positive, respectively.

Using w' and X we construct an auxiliary network N' from the original network as follows:

i) N' has an edge (i, j) with token $y_i - y_j + m_{ij}$ for each original edge (i, j) with $x_{ij} < c_{ij}$.

ii) N' has an edge (j, i) with token $y_j - y_i - m_{ij}$ for each original edge (i, j) with $x_{ij} > 0$.

iii) Add a new node s and new edges (s, i) with zero token for every wet node i.

For example, for the network in Figure 4.1 with flows as shown in Figure 4.2(a) and the following y_i's,

$$Y = [0\ 1\ 2\ 1\ 0\ 0]$$

the corresponding auxiliary network with node demands and edge tokens is shown in Figure 4.2(b).

To update Y, we apply the single-source shortest path algorithm (Algorithm SHORTEST-PATH) of Section 3.2 to N'' and determine the shortest paths and distances

from s to all the nodes in N''. The new Y' is given by

$$y'_i = y_i + d'_{s,i}$$

where $d'_{s,i}$ is the length of a shortest path from s to i in N'.

It can be shown that the new vector Y' and X satisfy equation (4.3). In other words, the update of Y has been achieved without violating the complementary slackness conditions.

4.1.3 Updating the Flow Vector X

Given X and Y', we now show how X can be updated to a new X' such that both X' and Y' satisfy complementary slackness conditions.

Our aim really is to update X so that we make as much progress as possible towards satisfying (4.1), namely, the equation $A\ X = W$. This requires that we push as much flow as possible from the current wet nodes towards the dry nodes. But this should be accomplished without violating (4.2) and (4.3). Thus, we can modify the flows only on those edges (i, j) for which $y'_i - y'_j + m_{ij} = 0$. Interestingly, all the edges on a shortest path from s to node i satisfy this requirement. Thus while pushing the flows from the wet to the dry nodes, we should use only these edges.

This suggests the use of the following network N'' for modifying the X vector:

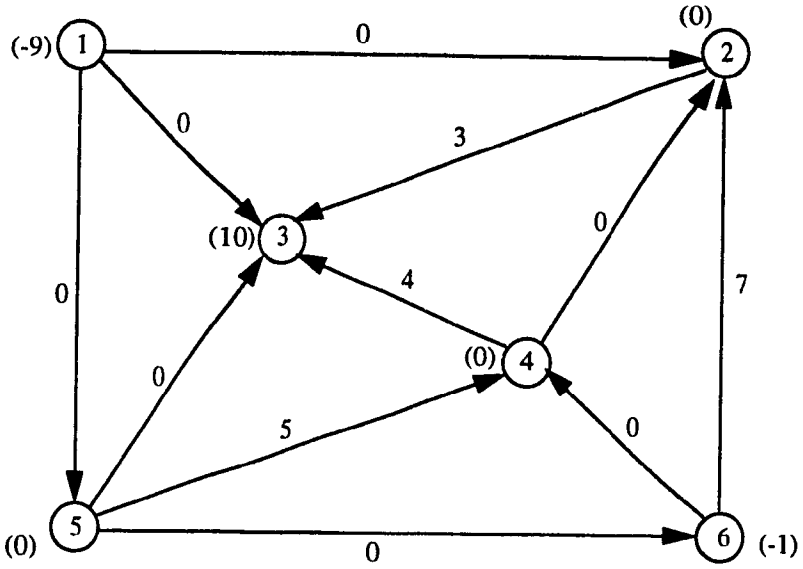
i) N'' is a subnetwork of N'.

ii) N'' has an edge (i, j) of capacity $c_{ij} - x_{ij}$ for each original edge (i, j) for which $y_i - y_j + m_{ij} = 0$ and $x_{ij} < c_{ij}$.

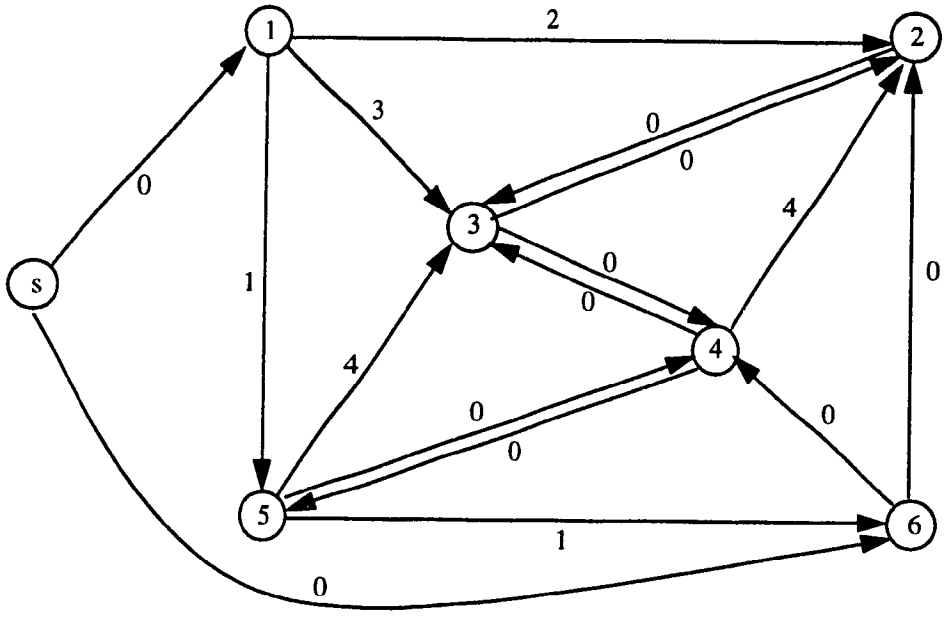
iii) N'' has an edge (j, i) of capacity x_{ij} for each original edge (i, j) for which $y_i - y_j + m_{ij} = 0$ and $x_{ij} > 0$.

iv) For each wet node i, N'' has edge (s, i) with capacity $-w'_i$.

v) N'' has a new node t and an edge (i, t) of capacity w_i for each dry node i.



(a) Network N of Figure 4.1 with a Flow and New Demands



(b) Auxiliary Network N' with Edge Tokens

Figure 4.2 Construction of Auxiliary Network

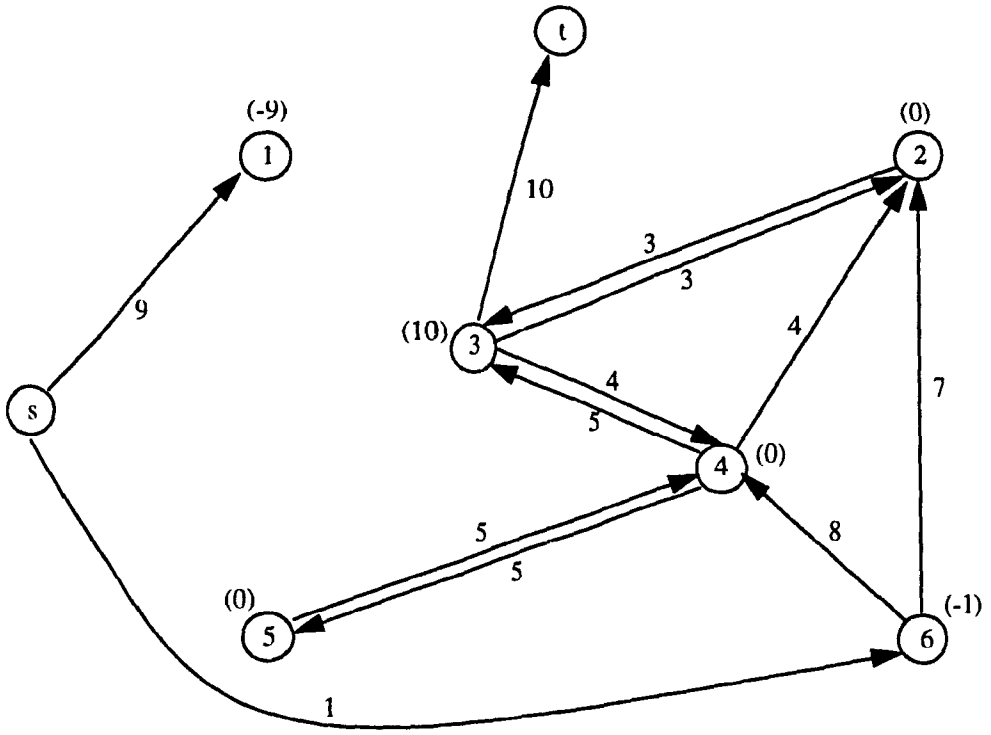


Figure 4.3 Network N'' constructed from Network N' of Figure 4.2 with given Flows and y values.

Thus pushing as much as possible from the wet nodes to the dry nodes reduces to pushing a maximum flow from s to t in N''.

For our example, the network N'' constructed from N' of Figure 4.2 is shown in Figure 4.3. Here the edge capacities are shown next to the edges.

At the completion of the maximum flow algorithm, we update the flow vector X to a new vector X' as follows.

For each (i, j) in the original network, let x_{ij}'' and x_{ji}'' be the corresponding flows in N''. Then the new flow x_{ij}' of the vector X' is given by

$$x_{ij}' = x_{ij} + x_{ij}'' - x_{ji}''$$

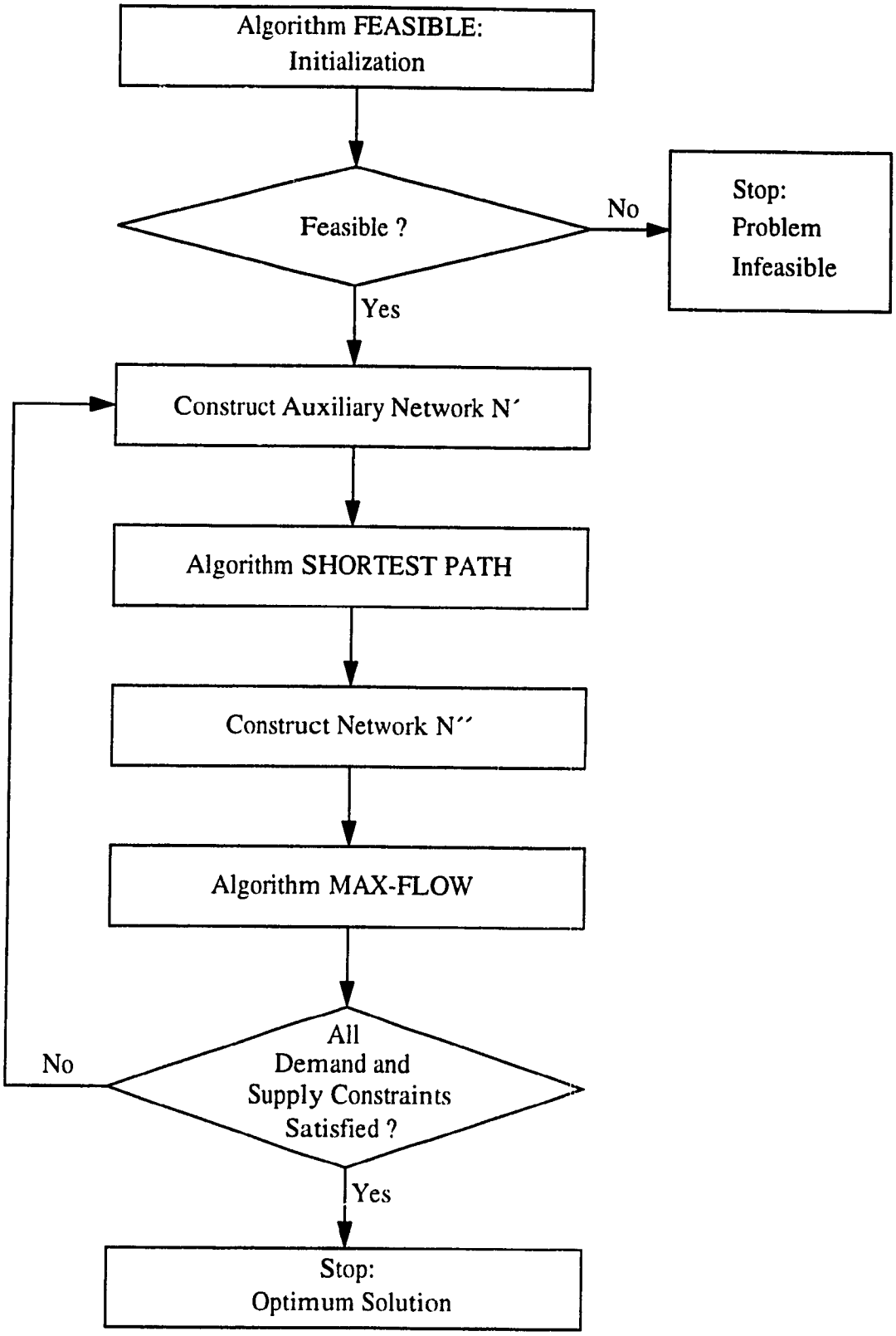


Figure 4.4 Primal-Dual Method

Note that some of the edges in N'' may not lie on any path from s to t. These edges would not play any role while performing the maximum flow algorithm. So, for an efficient implementation, it will be worthwhile to remove these edges before applying the maximum flow algorithm.

The network primal-dual method described above is summarized in Figure 4.4. Clearly, the primal dual method involves the three basic algorithms already discussed in the previous chapter. We now give a formal presentation of a parallel version of the primal dual method.

The following data structures are used in this algorithm.

outNodes	:	an array of sets. *outNodes [i]* is a set containing all the outnodes at node i in the original graph G.
inNodes	:	an array of sets. *inNodes [i]* is a set containing all the innodes at node i in the original graph G.
activeOutNodes	:	an array of sets. *activeOutNodes [i]* is a set containing all the outnodes at node i in the current graph G'.
activeInNodes	:	an array of sets. *activeInNodes [i]* is a set containing all the innodes at node i in the current graph G'.
nodeWeight	:	an array of integers. *nodeWeight [i]* indicates the original supply or demand at node i.
demand	:	an array of integers. *demand [i]* indicates the current supply or demand at node i.
demandZero	:	an array of boolean. *demandZero [i]* = TRUE indicates the condition *demand [i]* = 0 for the node i. It is initialized to FALSE.

firing_no : an array of integers. At any time, *firing_no [i]* represents the current firing number (dual variable) of node *i*.

token : a 2-dimensional array of integers. *token [i] [j]* represents the value of token of the edge from node *i* to node *j*, if there is an edge *(i, j)* in the original graph G; otherwise it contains a special value INFINITY outside the range of edge tokens.

activeToken : a 2-dimensional array of integers. *activeToken [i] [j]* represents the value of residual token for the edge *(i, j)*, if there is edge *(i, j)* in the current contracted graph G′; otherwise it contains a special value INFINITY outside the range of edge tokens. The variable *activeToken [i] [j]* is initialized to *token [i] [j]*.

capacity : a 2-dimensional array of integers. *capacity [i] [j]* indicates the capacity of the edge *(i, j)* if *(i, j)* is in the original graph G; otherwise it is equal to zero.

activeCapacity : a 2-dimensional array of integers. *activeCapacity [i] [j]* indicates the capacity of the edge *(i, j)* if *(i, j)* is in the current graph *G′*; otherwise it is equal to zero.

flowOut : a 2-dimensional array of integers. *flowOut [i] [j]* indicates the flow at node *i* along the edge *(i, j)*, if *j* is an outnode at *i*. Each *flowOut [i] [j]* is initialized to zero.

flowIn : a 2-dimensional array of integers. *flowIn [i] [j]* indicates the flow at node *i* along the edge *(j, i)*, if *j* is an innode at *i*. *flowIn [i] [j]* is equal to *–flowOut [j] [i]*.

Parallel Network Primal Dual Algorithm

```
procedure pd_initialization ( i )
{ Initialization of different variables assuming the input is read into the
corresponding inNodes, outNodes, token, capacity, and nodeWeight }
    terminate ← n - 2; { SOURCE and SINK don't have to do any work }
    activeOutNodes [i] ← ∅;
    activeOutNodes [i] ← ∅;
    demandZero [i] ← FALSE;
end procedure { pd_initialization }
procedure initialize_flow ( i )
    for all j in outNodes [i] do
        if (firing_no [i] + token [i] [j] ≥ firing_no [j])
        then
            flowOut [i] [j] ← 0;
            flowIn [j] [i] ← 0;
        else
            flowOut [i] [j] ← capacity [i] [j];
            flowIn [j] [i] ← – flowOut [i] [j];
        end if
    end for
end procedure { initialize_flow }
procedure initialize_demand ( i )
    demand [i] ← nodeWeight [i];
```

```
        for all j in outNodes [i] do
            demand [i] ← demand [i] + flowOut [i] [j]
        end for;
        for all k in inNodes [i] do
            demand [i] ← demand [i] + flowIn [i] [k]
        end for;
        if (demand [i] = 0 and demandZero [i] = FALSE)
        then
            lock terminate ← terminate - 1;
            demandZero [i] ← TRUE;
        end if;
    end procedure { initialize_demand }
    procedure create_active_neighbor_list ( i )
        for all j in outNodes [i] do
            if (flowOut [i] [j] < capacity [i] [j])
            then
                lock add j to activeOutNodes [i];
                lock add i to activeInNodes [j];
                activeToken [i] [j] ← firing_no [i] – firing_no [j] + token [i] [j];
            end if;
            if (flowOut [i] [j] > 0)
            then
                lock add j to activeInNodes [i];
```

```
            lock add i to activeOutNodes [k];
            activeToken [j] [i] ← firing_no [j] – firing_no [i] - token [i] [j];
        end if;
    end for;
    if (demand [i] < 0)
    then
        lock add i to activeOutNodes [SOURCE];
        lock add SOURCE to activeInNodes [i];
        activeToken [SOURCE] [i] ← 0;
    end if;
end procedure { create_active_neighbor_list }
procedure mark_nodes_and_update_firing_no ( i )
    activeOutNodes [i] ← ∅;
    activeInNodes [i] ← ∅;
    if (i ≠ SOURCE and i ≠ SINK)
    then
{ distance [i] represents token of a shortest path }
        firing_no [i] ← firing_no [i] + distance [i];
        if (demand [i] > 0)
        then
            mark [i] ← TRUE;
            while (pred [i] ≠ i and not mark [pred [i]]) do
                i ← pred [i];
```

```
            mark [i] ← TRUE;
        end while;
    end if
end if
end procedure { mark_nodes_and_update_firing_no )
procedure update_active_edges ( i )
    if (mark [i])
    then
        for all j in outNodes [i] do
            if (mark [j] and firing_no [j] = firing_no [i] + token [i] [j])
            then
                if (flowOut [i] [j] < capacity [i] [j])
                then
                    add j to activeOutNodes [i];
                    countOut [i] ← countOut [i] + 1;
                    activeCapacity [i] [j] ← capacity [i] [j] − flowOut [i] [j];
                end if;
                if (flowOut [i] [j] > 0)
                then
                    add j to activeInNodes [i];
                    countIn [i] ← countIn [i] + 1;
                    activeCapacity [i] [j] ← flowIn [i] [j];
                end if;
```

```
            end if
         end for;
      end if
end procedure { update_active_edges }
procedure create_antisymmetric_edges ( i )
      if (mark [i])
      then
         for all j in outNodes [i] do
            if (mark [j] and firing_no [j] = firing_no [i] + token [i] [j])
            then
               if (flowOut [i] [j] < capacity [i] [j])
               then
                  add j to activeOutNodes [i];
               end if;
               if (flowOut [i] [j] > 0)
               then
                  add j to activeInNodes [i];
                  activeCapacity [i] [j] ← flowIn [i] [j];
               end if;
            end if
         end for;
      end if
end procedure { create_antisymmetric_edges }
```

```
procedure update_source_and_sink_edges (i)
    s ← SOURCE;
    t ← SINK;
    if (mark [i])
    then
    if (demand [i] < 0)
    then
        add i to activeOutNodes [s];
        add s to activeInNodes [i];
        activeCapacity [s] [i] ← – demand [i];
    elsif (demand [i] > 0)
    then
        add t to activeOutNodes [i];
        add i to activeInNodes [t];
        activeCapacity [i] [t] ← demand [i];
    end if
end procedure { update_source_and_sink_edges }
procedure update_new_flow ( i )
    num ← 0;
    for all j in activeOutNodes [i] do
        num ← num + 1;
        if (num ≤ countOut [i])
        then
```

```
            flowOut [i] [j] ← flowOut [i] [j] + outFlow [i] [j];
            flowIn [j] [i] ← – flowOut [i] [j];
        end if;
    end for;
    num ← 0;
    for all j in activeInNodes [i] do
        num ← num + 1;
        if (num ≤ countIn [i])
        then
            flowOut [i] [j] ← flowOut [i] [j] + inFlow [i] [j];
            flowIn [j] [i] ← – flowOut [i] [j];
        end if;
    end for;
end procedure { update_new_flow }
procedure update_new_demand ( i )
    for all j in outNodes [i] do
        demand [i] ← demand [i] + flowOut [i] [j]
    end for;
    for all k in inNodes [i] do
        demand [i] ← demand [i] + flowIn [i] [k]
    end for;
    if (demand [i] = 0 and demandZero [i] = FALSE)
    then
```

```
            lock terminate ← terminate - 1;
            demandZero [i] ← TRUE;
        end if;
    end procedure { update_new_demand }
    procedure primal_dual ()
    { main procedure }
        for i ← 0 to n - 1 do in parallel
            pd_initialization ( i )
        end for;
        feasible ();
        for i ← 0 to n - 1 do in parallel
            if (i ≠ SOURCE and i ≠ SINK)
            then initialize_flow ( i )
            end if
        end for;
        for i ← 0 to n - 1 do in parallel
            if (i ≠ SOURCE and i ≠ SINK)
            then initialize_demand ( i )
            end if
        end for;
        while (terminate > 0) do
            for i ← 0 to n - 1 do in parallel
                if (i ≠ SOURCE and i ≠ SINK)
```

```
    then create_active_neighbor_list ( i )
    end if
end for;
shortest_path ();
for i ← 0 to n - 1 do in parallel
    mark_nodes_and_update_firing_no ( i )
end for;
for i ← 0 to n - 1 do in parallel
    if (i ≠ SOURCE and i ≠ SINK)
    then update_active_edges ( i )
    end if
end for;
for i ← 0 to n - 1 do in parallel
    if (i ≠ SOURCE and i ≠ SINK)
    then create_antisymmetric_edges ( i )
    end if
end for;
for i ← 0 to n - 1 do in parallel
    if (i ≠ SOURCE and i ≠ SINK)
    then update_source_and_sink_edges ( i )
    end if
end for;
max_flow ();
```

```
        for i ← 0 to n - 1 do in parallel
            if (i ≠ SOURCE and i ≠ SINK)
            then update_new_flow ( i )
            end if
        end for;
        for i ← 0 to n - 1 do in parallel
            if (i ≠ SOURCE and i ≠ SINK)
            then update_new_demand ( i )
            end if
        end for;
    end while;
end procedure { primal_dual }
```

4.2 Summary

In this chapter, we have discussed the details of the primal-dual approach to the transshipment problem as well as its parallel implementation. The network primal-dual approach involves repeated applications of the max-flow and shortest path algorithms discussed in the previous chapter. We have shown that Algorithm FEASIBLE can be adapted for initialization of the primal-dual method. This approach to initialization does not require constructing an auxiliary graph unlike the traditional approach to primal-dual initialization.

CHAPTER 5

PARALLEL NETWORK DUAL SIMPLEX METHOD

In this chapter, we develop a new approach to the solution of the dual transshipment problem and discuss its parallel implementation. This approach, called the Modified Network Dual Simplex method, is based on the traditional network dual simplex method [39] and uses concepts and results from the theory of marked graphs [43], [162]. As we shall see, this approach has several features which make it amenable for parallelisation.

The Modified Network Dual Simplex (MNDS) method for the solution of the dual transshipment problem involves repeated applications of three basic steps, namely, testing feasibility, shortest-path computations and performing concurrent pivot operations. In the following sections, we will present the essential features of this new approach and details of its implementation in a shared-memory programming environment.

5.1 Network Dual Simplex Method

Given a directed graph G = (V, E), let A, M and W be defined as in Section 2.1. Then we recall (Section 2.2) that the dual transshipment problem (DTP) is a linear program defined as follows.

Maximize: $W^t Y$

subject to

$$A^t Y \geq -M^t \tag{5.1}$$

$$Y \geq 0, \tag{5.2}$$

where Y is the column vector of node firing numbers. Thus, in DTP, we seek to find Y which maximizes $W^t Y$.

The network dual simplex method is essentially the simplex method of linear pro-

gramming 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 dual simplex method on this network. This step detects infeasibility of the DTP, whenever that is the case.

2 Perform a dual pivot operation (See Section 2.4). If a dual pivot operation is permissible, then it results in an improved value for the objective WY.

3 Check the new basic feasible solution for optimality. (The solution is optimal if the corresponding basic feasible tree permits no dual pivot operation.) If the solution 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 WY. To avoid cycling, Bland's anti-cycling rule [39] can be used in step (2) while selecting a pivot operation. Consult [39] for the details of the network dual simplex method.

5.2 Parallel Network Dual Simplex Method

As can be noted from the outline of the network dual simplex method presented in the previous section, 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.

5.3 Modified Network Dual Simplex Method

1 Test feasibility of the DTP (Apply Algorithm FEASIBLE). If feasible, construct a feasible solution.

2 Given a feasible solution Y, construct a basic feasible solution Y′ with WY′ ≥ WY

3 Check the optimality of the basic feasible solution Y′ obtained in step 2. 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).

4 Repeat steps 2 and 3.

Note that the above 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.

The first step in the solution of the dual transshipment problem is to test the feasibility of the problem. Suppose that the given dual transshipment problem is feasible. Then, after an application of Algorithm FEASIBLE (explained in Chapter 3), all the residual tokens associated with the edges will be non-negative. Let the corresponding graph be G'.

From this point onwards, our approach is to increase the value of the objective WY as much as possible until optimality is reached. This is achieved by firing the nodes in an appropriate manner without ever allowing the tokens to become negative. Thus we fire only positive weight nodes. This is repeated until no further firing of these nodes is possible. Let the graph at this point be denoted as G´´. Note that in G´´ at each positive weight node, there will be at least one edge with zero token, incident into the node.

Implementing the above step as described may result in redundant firings. We adopt the following strategy to avoid redundant firings.

Interestingly, we show that while transforming G´ to G´´, node i in G´ would be fired exactly f_i times where f_i is the token in G´ of a shortest path to i from a non-positive weight node. So to avoid redundant firings we compute in G´ the value of f_i's for all the positive weight nodes and then fire the nodes accordingly. This will transform the graph G´ to G´´. Note that this step involves only shortest path computations and can be done very efficiently using the parallel algorithm SHORTEST-PATH presented in Chapter 3.

Consider now the graph G´´. A number of edge tokens in G´´ will be zero. As mentioned above, no positive weight node can be fired in G´ because each such node will have an incident edge with zero residual token. So, at this point the only way we could increase the objective WY is to identify clusters of nodes in G´ with the following properties and fire them.

i) The weight of the cluster (the sum of the weights of the nodes in the cluster) is positive.

ii) For every node i in the cluster, there is no incident edge (j, i) with zero residual token, and with node j out side the cluster.

To identify such clusters, we proceed as follows.

First, we identify the subgraphs of G´ induced by those edges with zero residual

tokens. This subgraph may not be connected. In that case, let the connected components of this subgraph be G_1'', G_2'', ..., G_k''. Among these connected components, those with positive weight are precisely the clusters which satisfy the properties mentioned above and their firings would increase the value of the objective WY. Again, to carry out this in an efficient manner, we proceed as follows.

First, we construct a graph G''' as follows. Node i in G''' represents G_i'' and the edge e_{ij}''' (directed from node i to node j) will be assigned the smallest of the residual tokens of all edges directed from G_i'' to G_j''. Next we compute the weight of each node (which is now a cluster of nodes) in G''' given by the sum of the weights of all the nodes in the corresponding cluster. Then we apply the algorithm SHORTEST-PATH to compute for each positive weight node i, a shortest path from a non-positive weight node to i and then fire these nodes by the appropriate amounts. Note that firing a cluster x times results in adding x to the current firing numbers of all the nodes in the cluster.

We repeat the above process until all nodes coalesce into a single cluster. In other words, the edges with zero residual tokens would span all the nodes in G. At this point we can construct a *basic feasible solution* (represented by a 0-token spanning tree) of the dual simplex. We now test the optimality of the solution using the simplex optimality criterion. Suppose the solution is not optimal. Then we determine for each branch *(i, j)* of the spanning tree (representing the basic solution) the corresponding fundamental cutset. Let the corresponding node partition be $(V_i, \overline{V}_i)$. Assume without loss of generality that the node i is in V_i. Then recall that V_i is the fundamental cluster corresponding to the branch *(i, j)*. If the weight of the cluster $\overline{V}_i$ is positive then we can fire $\overline{V}_i$ to increase the value of the objective. Again, note that firing a fundamental cluster $\overline{V}_i$ is in fact the same as a simplex pivot operation with respect to the branch *(i, j)*. Our intention is to fire all the positive weight fundamental clusters concurrently. In doing so, we encounter two problems. First, firing all positive weight fundamental clusters concurrently may result in producing nega-

tive residual tokens. Secondly, we need an efficient strategy to identify the positive fundamental clusters in the graph. Our strategy is as follows.

Instead of looking for positive fundamental clusters, our approach is to identify a fundamental cluster and fire that cluster appropriately. That is, if the cluster is positive weighted, then fire that cluster in a regular manner (using positive firing). On the other hand, if the fundamental cluster is negative weighted, then we use negative firing to fire that cluster.

The advantage of employing both forms of firing is that after we have computed a fundamental cluster, we do not have to discard it even if it is negative weighted. This would save a lot of computations and also the number of pulses required to traverse the tree. In fact, if we employ this strategy, then the number of pulses required to complete the concurrent pivot phase of our algorithm, will be half the length of a longest path in the basic feasible tree. The problem, however, is that we cannot employ both forms of firing in the same pulse. So we use two subpulses in each firing pulse. One is to fire in a positive way and the other to fire in a negative way. Some details of this approach are given next.

In each iteration of the concurrent pivot phase, only the pendant nodes (nodes with only one tree edge) will be active. Each of them checks their node weights and participates in the appropriate firing pulse. Then they traverse the tree along the only tree edge they have and merges with the node at the other end. This will result in a shorter tree with new pendant nodes/clusters. Therefore, the number of pendant nodes in each iteration varies dynamically, but we are guaranteed that there will be at least two pendant nodes in each iteration. So, the number of pulses required to traverse the complete tree will be at most half the length of a longest path in the tree.

The above step is repeated until the whole tree collapses into one node/cluster. Note, at any stage, the pendant nodes/clusters that are active are mutually exclusive (i.e., not overlapping with each other) and therefore can be fired and merged with the other

nodes concurrently.

If there is any non-zero firing at any stage of concurrent pivots, this means the solution is not basic any more. So, the algorithm will go back to the second step of clustering and computing shortest paths to find a new basic feasible solution. The objective value of this new solution will be greater than or equal to that of the previous one. This guarantees the progress of the algorithm and its eventual termination. If no firing takes place at any stage of the concurrent pivot phase, then the algorithm terminates and the solution will be optimum.

Since concurrent pivots employ both negative and positive firings, at optimalilty, the final y_i's while satisfying the constraint (2.3), may not satisfy the constraint (2.4). This can easily be corrected by reducing all the y_i's by an amount equal to the minimum of all y_i's. In other words, we perform the following:

$$k \leftarrow \min \{ y_i \}$$

$$y_i \leftarrow y_i - k, \text{ for all } i$$

Summarizing, the Modified Dual Simplex method consists of the following main steps. We start with the given graph and the associated edge tokens prescribed by M.

Step 1: We apply Algorithm FEASIBLE to test feasibility of the problem. At the end of this step, all the edge tokens will be non-negative.

Step 2: We then fire positive weight nodes as much as possible with a view to increasing the objective function. This process can be performed very efficiently using Algorithm SHORTEST-PATH. When no more firings of positive weight nodes are possible, the nodes will partition into clusters. At this point, we fire positive weight clusters as much as possible. Again, this can be done efficiently by constructing a smaller graph in which a node represents a cluster and then applying Algorithm SHORTEST-PATH on this new graph.

Step 3: We repeat Step 2 until we obtain a basic solution (represented by a spanning tree) of the dual simplex. At this point, we test the optimality of the solution using the simplex optimality criterion. If the solution is not optimal, we fire concurrently the fundamental clusters in an appropriate manner and increase the value of the objective as much as possible.

At the end of Step 3, the solution may not be basic. We now repeat steps 2 and 3 until optimality is reached.

The interesting features of the above approach are:

i) No auxiliary graph is constructed to test feasibility.

ii) Node and cluster firing operations of Step 2 can be performed efficiently using Algorithm SHORTEST-PATH.

iii) Several simplex pivot operations are performed concurrently in Step 3. The classical simplex approach does not permit concurrent pivots because these operations may destroy basicness of the solution.

iv) If the solution at the end of Step 3 is not basic, Step 2 will convert it to a basic solution with an objective value greater than or equal to that of the previous basic solution. In other words, Step 2 converts a non-basic solution to a basic one without ever decreasing the value of the objective function.

Figure 5.1 summarizes the different steps of MNDS method. A parallel implementation of the above algorithm and the resolution of several issues that we encounter are explained next. The graph G shown in Figure 5.2 will be used to illustrate each step of our algorithm.

5.4 Implementation of MNDS Method

As before each node is associated with a single processor and information is com-

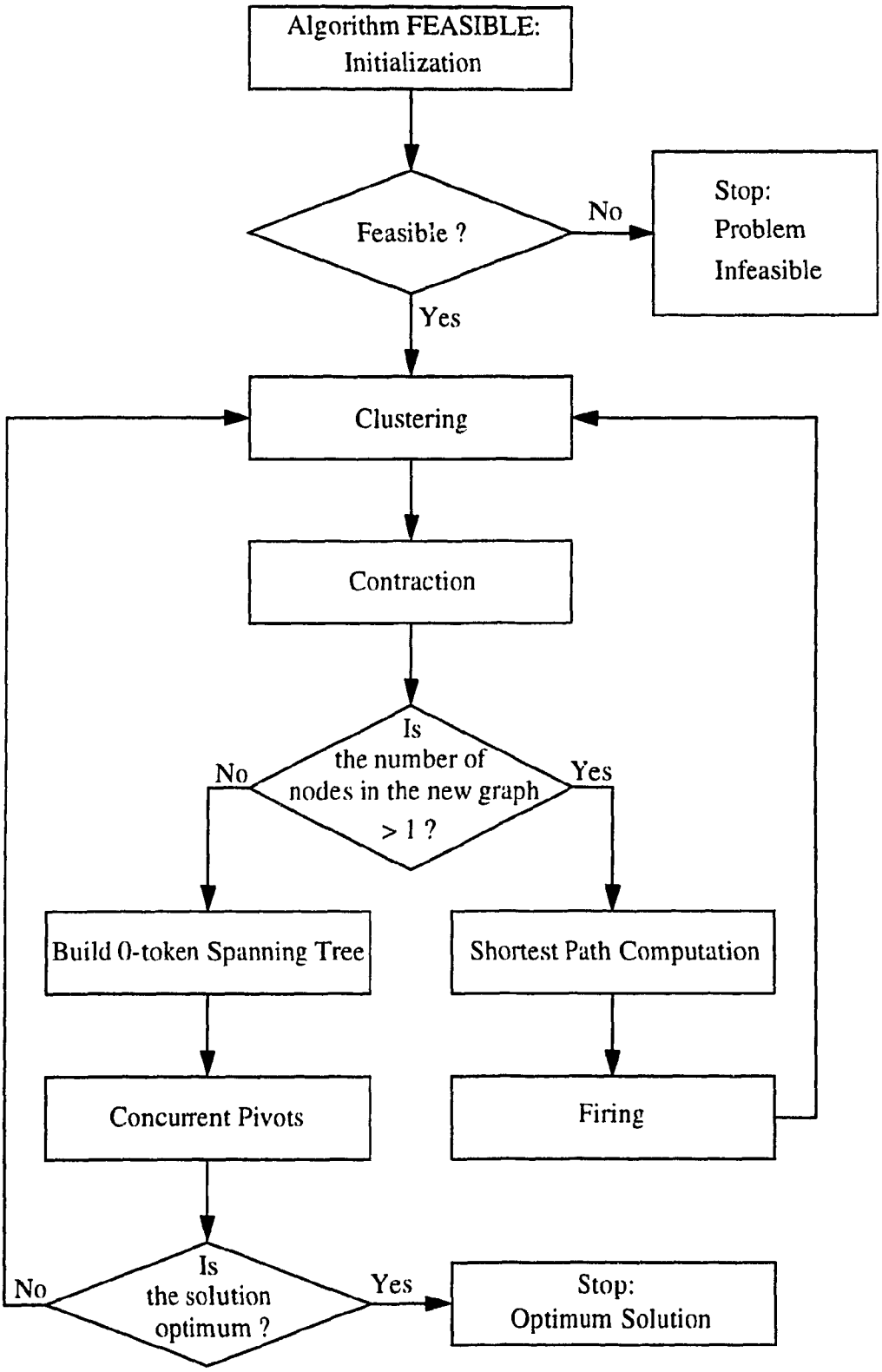
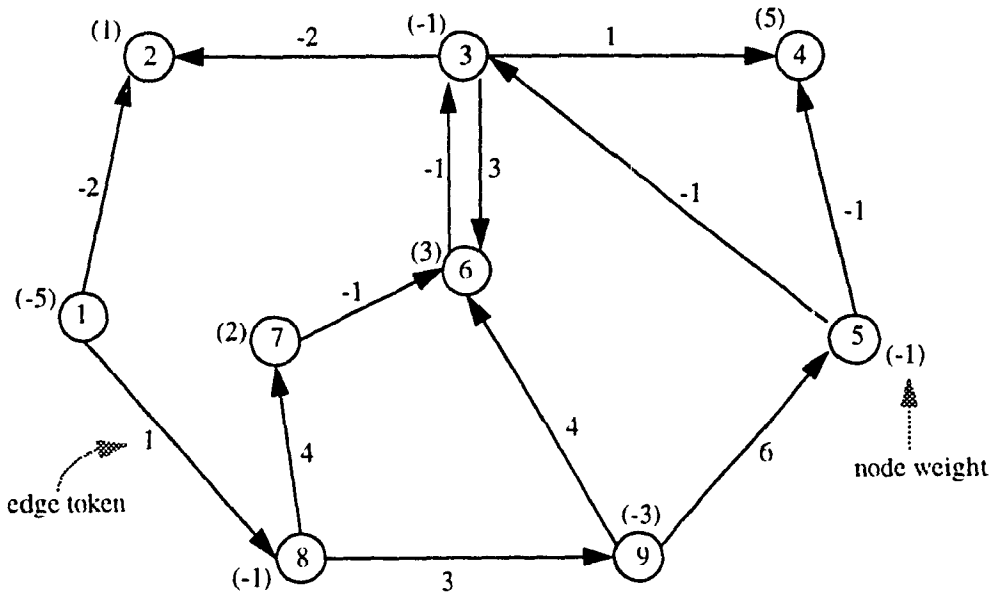


Figure 5.1 Modified Network Dual Simplex Method

municated from one processor to another through shared-memory.

5.4.1 Data Structures

The following data structures are used in the implementation of this algorithm. Data structures which are used only in certain procedures will be defined at the appropriate places.



Node numbers are shown inside nodes.
Node Weights and edge tokens are as shown.

Figure 5.2 An example to illustrate our algorithm: Given graph G.

n : an integer variable. It denotes the number of nodes in the original graph G.

activeN : an integer variable. It denotes the number of nodes in the current contracted graph G′.

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.

activeToken : a 2-dimensional array of integers. *activeToken [i] [j]* represents the value of residual token of the edge *(i, j)* in the current contracted graph G′. If there is no edge *(i, j)* in G′, *activeToken [i] [j]* contains a special value INFINITY outside the range of edge tokens. The variable *activeToken [i] [j]* is initialized to *token [i] [j]*.

inNodes : an array of sets. *inNodes [i]* is a set containing all the innodes at node *i*.

outNodes : an array of sets. *outNodes [i]* is a set containing all the outnodes at node *i*.

activeOutNodes : 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]*.

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

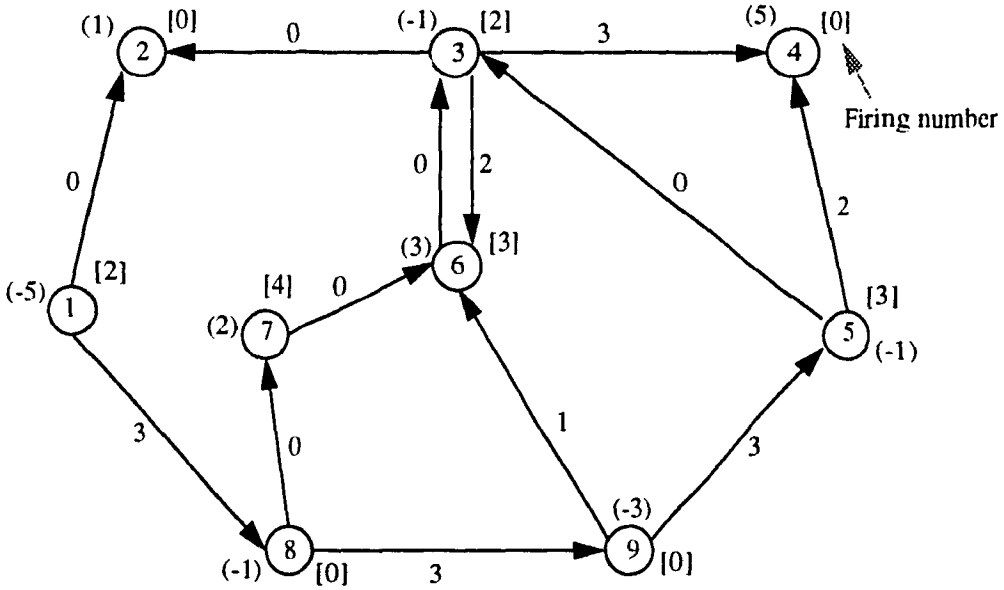
nodeWeight : an array of integers. *nodeWeight [i]* represents the weight of the node *i*.

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 containing node *i*.

source : an array of integers. The variable *source [i]* contains the root of the cluster in which node *i* is present. (The root of a cluster is a node in the cluster selected to represent this cluster. This term will be defined formally in a later section.)

zeroNodes : an array of sets. *zeroNodes [i]* contains all zero outnodes at node *i*.

treeNodes : an array of sets. The variable *treeNodes [i]* indicates the set of adjacent nodes (both incoming and outgoing) at node *i* such that the edges between *i* and each node *j* in the set *treeNodes [i]* are in the tree.



Firing numbers of nodes are shown in square brackets.

Figure 5.3 The graph G after feasibility testing.

5.4.2 Testing Feasibility of the Dual Transshipment Problem

Complete details of a parallel implementation of Algorithm FEASIBLE to test the feasibility of DTP are given in Section 3.1.

Figure 5.3 shows the graph G after applying Algorithm FEASIBLE on the graph G in Figure 5.2.

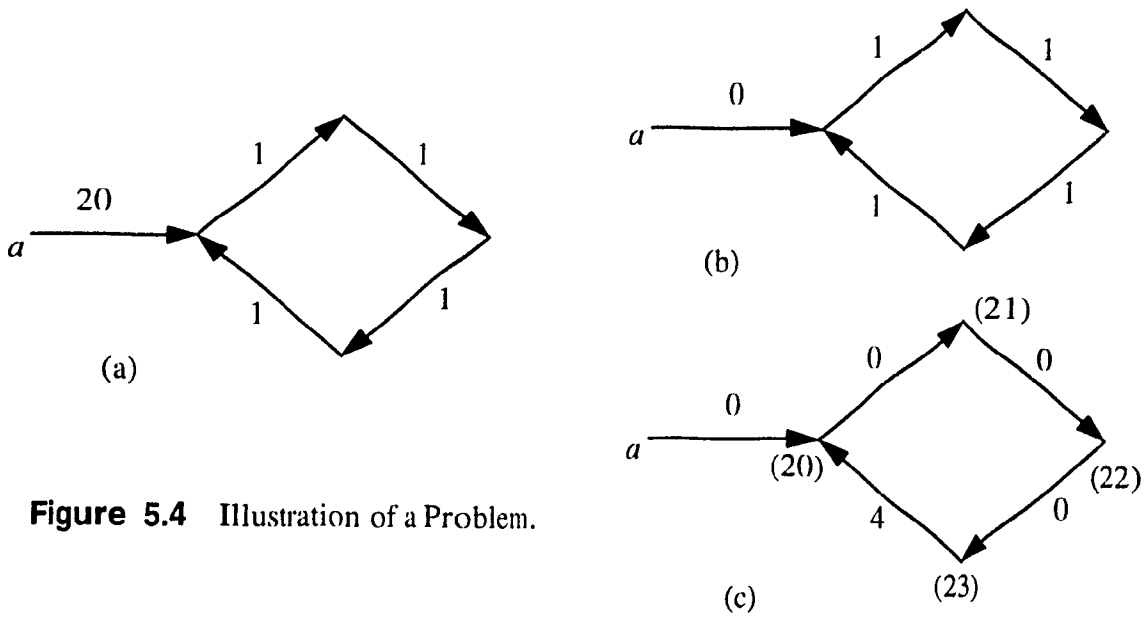


Figure 5.4 Illustration of a Problem.

5.4.3 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 positive weight nodes concurrently as much as possible. We then repeat these concurrent firings until no further firings are possible. Note that 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 Figure 5.4. During the first pulse of concurrent fir-

ings, all the nodes will be fired exactly once. After twenty pulses of these firings, the residual tokens will be as shown in Figure 5.4(b). After the 23rd pulse, the residual tokens will be as shown in Figure 5.4(c). The total number of firings of each node is also given in this figure.

On the other hand, suppose we first determine the tokens 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 would be fired as in Figure 5.4(c). This is not an accident. In fact we can prove the following.

Theorem 5.1

If there exists a directed path from i to j of token x, then the node j 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. 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 to node i.

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 residual 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 $S_1, \ldots, S_k$ such that each S_i with $|S_i| > 1$ has at least one non-positive weight node and all nodes reachable from this node by a directed path of zero residual token are also in S_i. If a node in S_i 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 may then be performed.

Our approach to construct a basic feasible solution from a given feasible solution involves repeated application 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 5.4.4.)

3 Shortest path computations and cluster firings.

Each one of these phases is discussed below.

5.4.3.1 Clustering

Clustering involves finding for each non-positive weighted node i, the set of all nodes reachable from i by directed paths of edges with zero residual tokens. If a positive weight node j is reachable from more than one non-positive weight 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 weight node in that cluster. In our description of the clustering process, we use the following additional data structures.

source : *source [i]* at the end of this phase represents the root of 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.

jobDone : an array of boolean. In the beginning, for each i, *jobDone [i]* is initialized to FALSE. As node i propagates *source [i]* to all of its zero outnodes, it will change *jobDone [i]* to TRUE.

The clustering phase has three subphases.

Subphase 1

In 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 *jobDone [i]* = FALSE, then node *i* sets *jobDone [i]* = TRUE and performs the following for each zero out-node 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 if no action is taken by any processor during a pulse. Though this subphase may contain several pulses, each node *i* will propagate its *source* value during only one pulse.

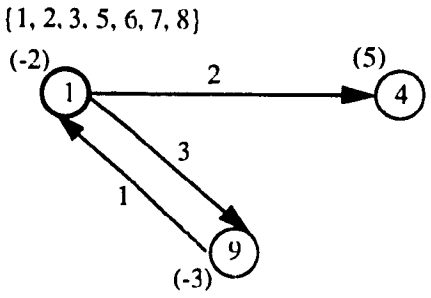
Note that at the end of the first subphase each non-positive node *i* will have *source [i]* ≤ *i*. It is likely that certain positive nodes have their source variables equal to NOTHING, unchanged from their initial values. For such nodes we set *source [i] = i* before initiating subphase 2.

Subphase 2

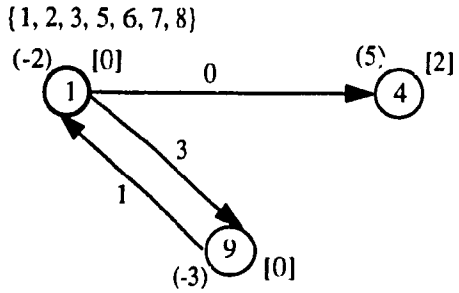
In this subphase only non-positive nodes will be active. During this subphase, each node i performs the following actions.

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.

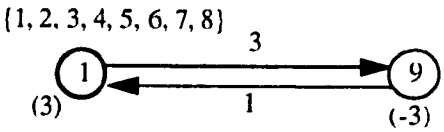
We can easily show that at the end of the second subphase, *source [i]* for each non-positive node will contain the least numbered non-positive node in its cluster.



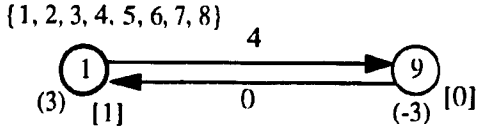
(a) The graph G′ after contracting G.



(b) Shortest path calculations and Firing of G′.



(c) The graph G′′ after contracting G′.



(d) Shortest path calculations and Firing of G′′.

Note: Each thick-lined node here represents the source of a cluster.
The corresponding set of nodes is indicated in braces.

Figure 5.5 Illustration of contraction and shortest path calculations for G of Figure 5.3.

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 common *source* value will also give the least-numbered non-positive node in that cluster if the cluster has cardinality greater than 1. In other words, *source [i]* is in fact the root of the cluster containing node *i*.

For our example, each thick-lined node in Figure 5.5 represents a cluster. The nodes in that cluster are given in braces. Nodes 4 and 9 will not join any cluster and therefore the source of node 4 is NOTHING while the source of node 9 is itself. Node 1 is the source of the cluster {1, 2, 3, 5, 6, 7, 8} (See Figure 5.5(a)).

This completes our discussion of the clustering phase.

A formal presentation of a parallel version of the above algorithm for clustering is given next.

```
procedure get_last_link ( i )
{ travels through the chain of sources at node i and returns the source at the end
of this chain }
    t ← i;
    s ← source [t];
    while (s ≠ t) do
        t ← s;
        s ← source [t];
    end while;
    return s;
end procedure {get_last_link}
```

```
procedure change_links (n1, n2)
{ Two chains of sources at n1 and n2 are traversed and linked together with the
smallest source in these chains }
        s1 ← get_last_link (n1);
        s2 ← get_last_link (n2);
        if (s1 < s2)
        then
            s ← s1;
            source [s2] ← s1;
        else
            s ← s2;
            source [s1] ← s2;
        end if;
    return s;
end procedure {change_links}
procedure propogate_source ( i )
{ The node i will propagate its source information to each of its zero outnodes. If
there is more than one source at any node, then these sources are linked and the
least numbered one will be propagated further. }
    if (not jobDone [i] and source [i] ≠ NOTHING)
    then
        if (zeroNodes [i] ≠ ∅)
        then
            notFinished ← TRUE;
```

```
            s ← source [i];
            for all j in zeroNodes [i] do
               lock source [j];
                  if (source [j] ≠ NOTHING)
                  then s ← change_links (s, source [j])
                  else source [j] ← s
                  end if;
               unlock source [j];
            end for;
         end if;
         jobDone [i] ← TRUE;
      end if;
end procedure {propagate_source}
procedure find_negative_source ( i )
{ Each non-positive node i will find its root }
      if (clusterWeight [i] ≤ 0)
      then
         source [i] ← get_last_link ( i );
         if (i = source [i])
         then no_of_nodes ← no_of_nodes + 1
         end if;
      end if
end procedure {find_negative_source}
```

```
procedure find_positive_source ( i )
{ Each positive node i will find its root }
    if (clusterWeight [i] > 0)
    then source [i] ← get_last_link ( i )
    end if
end procedure {find_positive_source}
procedure clustering ()
{ main algorithm }
    notFinished ← TRUE;
    for i ← 0 to N - 1 do in parallel { initialization }
        jobDone [i] ← FALSE;
        if (clusterWeight [i] ≤ 0)
        then source [i] ← i
        else source [i] ← NOTHING
        end if;
    end for;
    while (notFinished) do
        notFinished ← FALSE;
        for i ← 0 to N - 1 do in parallel
            propagate_source ( i )
        end for;
    end while;
    for i ← 0 to N - 1 do in parallel
```

```
        find_negative_source ( i )
    end for;
    for i ← 0 to N - 1 do in parallel
        find_positive_source ( i )
    end for;
end procedure { clustering }
```

5.4.3.2 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]*, $j \neq i$. 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.

Now we proceed to present 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 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]* $\neq i$.

Since there may be more than one node trying to access the root's *clusterWeight* variable at the same time, the *clusterWeight* variable of the root is locked before it is updated. After the update, the node will unlock this variable so that another node can update it. Note the locking and unlocking of each root'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 node *j* in *activeOutNodes [i]*. Suppose *k* is the root of the node *i* and *l* is the root of the node *j*. Then node *i* takes the following actions.

1 If $k = l$, then remove *j* from *activeOutNodes [i]*.

2 If $k \neq l$ and $l \neq j$, then do the following:

   a) remove *j* from *activeOutNodes [i]*.

   b) add *l* to *activeOutNodes [i]*.

   c) If *activeTokens [i] [j]* < *activeTokens [i] [l]*, then write *activeTokens [i] [l] = activeTokens [i] [j]*.

At the end of this step, for each node *i* and each node $j \in$ *activeOutNodes [i]*, an edge (*i*, *l*) will be created where *l* = *source [j]*, whenever *source [j]* $\neq$ *source [i]*. The residual token of this edge is the minimum token of all the edges between *i* and all *j* such that *l* = *source [j]*.

In the second step, each node *i* 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 to access their root's data structures. Therefore locking and unlocking of root's data structure is necessary in this step. But the data structure of each root 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.

The formal presentation of the algorithm is given next.

```
procedure process_edges ( i )
{ Each node i will process its active outnodes and change each outnode to the root
of the cluster in which the outnode is present. If there is more than one edge to a root
then it will keep the most constraining edge-token as the new token. }
    for all j in activeOutNodes [i] do
        if (source [j] = source [i])
        then delete j from activeOutNodes [i]
        elsif (source [j] ≠ j)
        then
            if (token [i] [j] < token [i] [source [j]])
            then token [i] [source [j]] ← token [i] [j]
            end if;
            delete j from activeOutNodes [i];
```

```
            add source [j] to activeOutNodes [i];
          end if
        end for
end procedure {process_edges}
procedure merge_edges ( i )
{ In this procedure, each node i will merge its active outnodes set and its edges
with that of its root, if the root of the cluster is not itself. }
      for all j in activeOutNodes [i] do
          lock source [i];
            if (token [i] [j] < token [source [i]] [j])
            then token [source [i]] [j] ← token [i] [j]
            end if;
          unlock source [i];
      end for;
      lock activeOutNodes [source[i]] ← activeOutNodes [source[i]] ∪
                                                    activeOutNodes [i];
end procedure {merge_edges}
procedure contraction ()
{ main algorithm }
      for i ← 0 to N - 1 do in parallel
          process_edges ( i )
      end for;
      for i ← 0 to N - 1 do in parallel
```

```
        s ← source [i];
        if (i ≠ s)
        then
            lock clusterWeight [s] = clusterWeight [s] + clusterWeight [i];
            merge_edges (i);
        end if;
    end for;
end procedure { contraction }
```

5.4.3.3 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 to it 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 using the algorithm SHORTEST-PATH explained in Chapter 3. 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]* = ∞, where *distance [i]* refers to the token of a shortest path *i* from a non-positive node. This indicates unboundedness of the DTP because the node *i* can be fired an unbounded number of times increasing the value of WY to an unbounded value. If this happens, our algorithm will terminate with an appropriate error message.

In the algorithm SHORTEST-PATH, all the sources will initialize their *newDistance* variables to zero while other nodes will initialize them to INFINITY. Thus, all non-positive weighted clusters will act as sources and initialize their *newDistance* variables to zero and all positive weighted clusters will initialize their *newDistance* variables to INFINITY. But it could so happen that the graph at an intermediate stage is made up entirely of zero weight nodes. This condition can be detected by checking if at least one *newDistance* variable in the graph is initialized to INFINITY. If not, then a special initialization routine is called to initialize the graph. The special initialization routine will allow only one node to initialize its *newDistance* variable to zero while all other nodes are allowed to initialize their *newDistance* variables to INFINITY. This node will be selected on a *first-come first-serve* basis, but it should have at least one outgoing edge. The formal details of this special initialization routine are given next.

```
procedure sp_special_initialize ( i )
{ This procedure is called if the variable notFinished is still FALSE after the
procedure sp_initialize is executed for all the active nodes. This happens only when
the entire graph is made up of zero weight nodes. One node with at least one
outgoing edge will be picked as source and initialized accordingly while all other
nodes are initialized as non-source nodes. }
    if (|activeOutNodes [i]| > 0)
    then
        lock if (notFinished)
            then newDistance [i] ← INFINITY
            else
                newDistance [i] ← 0;
                notFinished ← TRUE;
            end if
```

```
    else newDistance [j] ← INFINITY
end procedure { sp_special_initialize }
```

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. (Note that the edge tokens are the residual tokens that result after firing the nodes as specified by Y).

   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.

5.4.4 Building a 0-token Spanning 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 - both incoming and outgoing - adjacent to *i*. Maintaining such a set has not been necessary for any of the algorithms discussed thus far in this section; 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. We next explain the details of this algorithm.

The following data structures are used.

tree : a set containing all the nodes that have joined the tree till that time. Initially it is a null set.

treeNodes : an array of sets. *treeNodes [i]* indicates the set of nodes (both incoming and outgoing) adjacent to node *i* such that the edges between *i* and each node *j* in the set *treeNodes [i]* are in the tree. Each *treeNodes [i]* is initialized to the null set.

ROOT is a node chosen arbitrarily. We select the node numbered zero as ROOT.

The algorithm to build the 0-token tree is an iterative one. The algorithm terminates when *jobDone [i]* is equal to TRUE for all *i*. A node *i* can join the tree only if it is adjacent to at least one node in the tree. In other words, node *i* can join the tree only if $tree \cap zeroNodes[i] \neq \emptyset$. Locking and unlocking operations are used to make sure only one node can join the tree at a time. Initially, the *tree* is empty and so any node can join the tree by simply adding all of its *zeroNodes* to *tree*. Of course, locking and unlocking operations prevent more than one node seeing the *tree* empty and joining it simultaneously.

The following operations are performed at node *i*.

if ($zeroNodes[i] \cap tree \neq \emptyset$ **or** $i \in tree$)

then

$treeNodes[i] = treeNodes[i] \cup (zeroNodes[i] - tree)$

$tree = tree \cup zeroNodes[i]$

$tree = tree + i$

The algorithm terminates when all nodes have joined the tree. We can show very easily that the algorithm terminates eventually. Since this phase is entered only upon the successful termination of the clustering, contraction and shortest paths phases at the end of which all the nodes in the graph are collapsed into one single cluster, we can prove that there will be a zero-weighted (undirected) path between every pair of nodes and therefore a 0-token spanning tree can be built using only 0-token edges.

For our example, after performing the clustering and contraction on graph G'' in Figure 5.5(d), all the nodes coalesce into a single node. The graph G with node firing numbers (performed thus far) and the residual tokens is shown in Figure 5.6. At this point a 0-token spanning tree is available. This is shown in Figure 5.7.

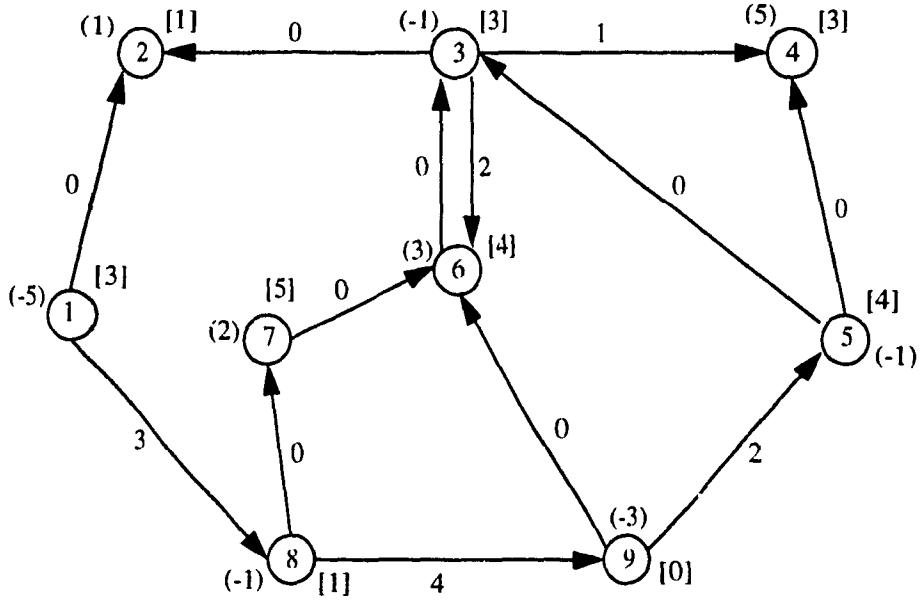


Figure 5.6 Graph G with its residual tokens at the end of clustering and shortest path calculations phase.

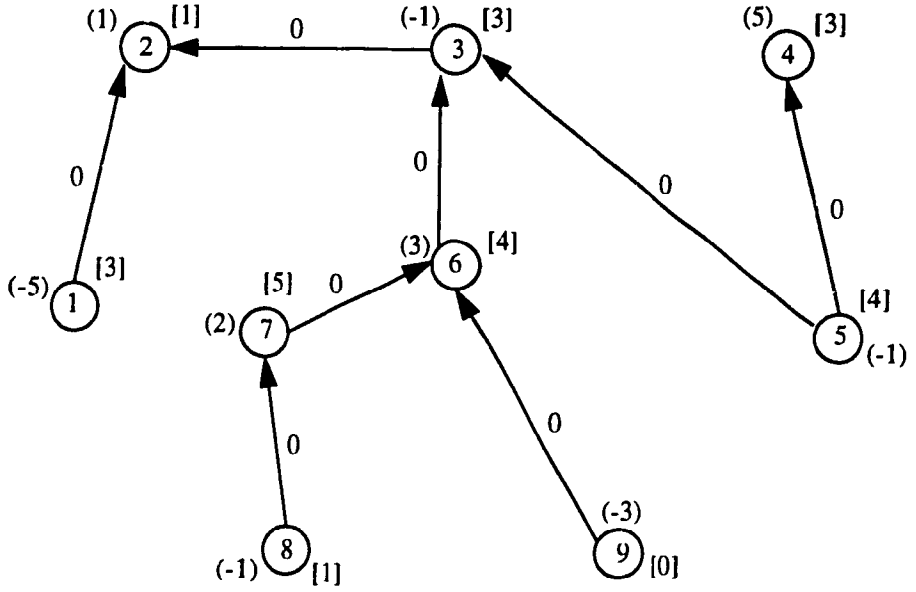


Figure 5.7 A 0-token spanning tree of the graph G in Figure 5.6.

The formal algorithm is presented next.

```
procedure initialize_tree ( i )
    calculate new residual tokens ;
    calculate new zero outnodes ;
    jobDone [i] ← FALSE ;
    treeNodes [i] ← ∅;
    if (i = ROOT)
    then tree ← ∅
    end if;
end procedure { initialize_tree }
procedure inform_other_end (i, temp)
{ If an edge (i, j) is in the 0-token spanning tree, then inform the node j about it. }
    for all j in temp do
```

```
        treeNodes [j] ← treeNodes [j] + i
    end for
end procedure {inform_other_end}
procedure join_tree ( i )
{ Each node i joins the tree either if it is the first one to do so or if it or one of its
zero outnodes is already in the tree. It also calculates which of its zero outgoing
edges will be in the tree and informs the corresponding zero outnodes. }
    temp ← zeroNodes [i] + i ;
    if ( i = ROOT )
    then
        lock tree ← temp ;
        treeNodes [i] ← zeroNodes [i]
        inform_other_end (i, zeroNodes [i]) ;
        jobDone [i] ← TRUE ;
    elsif ( tree ∩ temp ≠ ∅ )
    then
        lock tree;
            tcopy ← tree ;
            tree ← tree ∪ temp ;
        unlock tree;
        temp ← zeroNodes [i] - tcopy ;
        if (i ∉ tcopy)
        then
```

```
            s ← zeroNodes [i] ∩ tcopy ;
            m ← get the first member in s ;
            temp ← temp + m ;
        end if ;
        inform_other_end (i, temp) ;
        treeNodes [i] ← treeNodes [i] ∪ temp;
        jobDone [i] ← TRUE ;
    end if;
end procedure { join_tree }
procedure build_tree ( )
{ main algorithm }
    for i ← 0 to N - 1 do in parallel
        initialize_tree ( i )
    end for;
    for i ← 0 to N - 1 do in parallel
        join_tree ( i )
    end for ;
end procedure {build_tree}
```

5.4.5 Concurrent Pivots

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

For each node *i*, *father [i]* will denote the unique father of *i* in a depth-first-search of the basic feasible tree T.

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

num_tree_edges : an array of integers. Initially, *num_tree_edges [i]* will denote the number of nodes that are adjacent to *i* in T.

new_num_tree_edges : Same as *num_tree_edges*. As the bottom-up traversal of T proceeds, the edges will be contracted and the tree will shrink dynamically. This dynamic change of tree edges in an iteration is indicated by *new_num_tree_edges*. At the end of each iteration, *new_num_tree_edges* is copied into *num_tree_edges*.

cluster_firing_no : an array of integers. *cluster_firing_no [source [i]]* will denote the maximum number of times the fundamental cluster which contains node *i* can be fired. This variable is initialized to INFINITY.

At the end of a cluster finding pulse, *source [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 *source [i]*.) and *clusterWeight [source [i]]* will denote the weight of this cluster. Initially, *clusterWeight [i] = nodeWeight [i]*.

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_edges [i]* $\leq$ 1 and

clusterWeight [i] > 0 will be active. Each such node i will process the nodes j in *treeNodes [i]* one by one and discard those edges (j, i) with *source [j] = source [i]*. From the remaining edges, it will calculate the minimum residual edge token, say, f_i. Note that f_i will be the minimum of the residual token on the incoming edges at i. After calculating f_i, node i will lock the *clusterFiringNumber* variable of *source [i]* and will write

clusterFiringNumber [source [i]] = f_i, if f_i < *clusterFiringNumber [source [i]]*

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

Negative Cluster Firing

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

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

Determining New Clusters

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 *new_num_tree_edges [i]* by 1

3 Decrement *new_num_tree_edges [father[i]]* by 1

4 Delete i from *treeNodes [father [i]]*

5 Write *source [i] = father [i].*

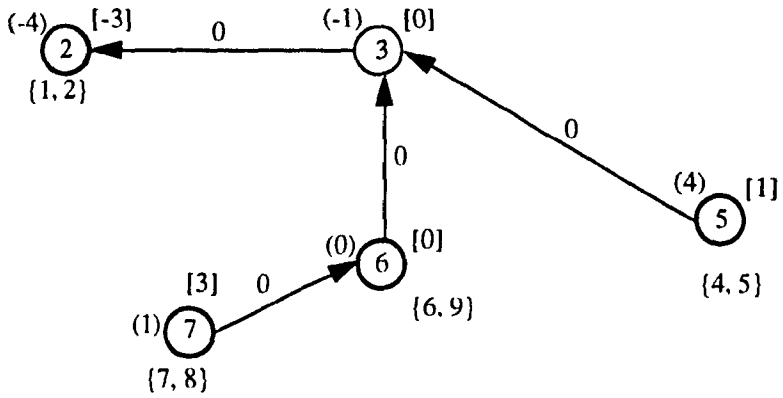
During the second pulse only nodes with *num_tree_edges [i]* ≤ 1 will be active. Each such node *i* will write

1 *source [i] = source [source [i]]*

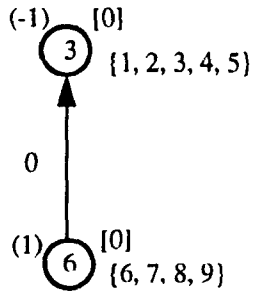
2 *num_tree_edges [i] = new_num_tree_edges [i]*

3 *clusterWeight [i] = clusterWeight [father [i]]*

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



(a) After one iteration.



(b) After two iterations.

Figure 5.8 The 0-token spanning tree at different iterations.

Figure 5.8 shows the zero token spanning tree at different points in the concurrent pivoting phase. After one iteration, the tree will be as shown in Figure 5.8(a). Here node 7

(representing the cluster {7, 8}) will fire 3 times while node 5 (representing the cluster {4, 5}) will fire once. Both these clusters fire concurrently. On the other hand, node 2 (representing the cluster {1, 2}) will negative fire 3 times. Then the tree will contract to as shown in Figure 5.8(b). After this pulse, the tree will coalesce into one cluster thus terminating the concurrent pivots phase. Then the algorithm will return to clustering and shortest path computation phase, but quickly terminate that phase without any improvement in the objecting value. One can build a new 0-token spanning tree and proceed to the concurrent pivot phase. Here again nodes 3, 4, 5, 6 and 9 together fire 3 times. The algorithm goes through the clustering and shortest path computations and then concurrent pivots again without any improvement in the objective value. At this point, the algorithm terminates. The final firing numbers and residual tokens of the graph G are shown in Figure 5.9.

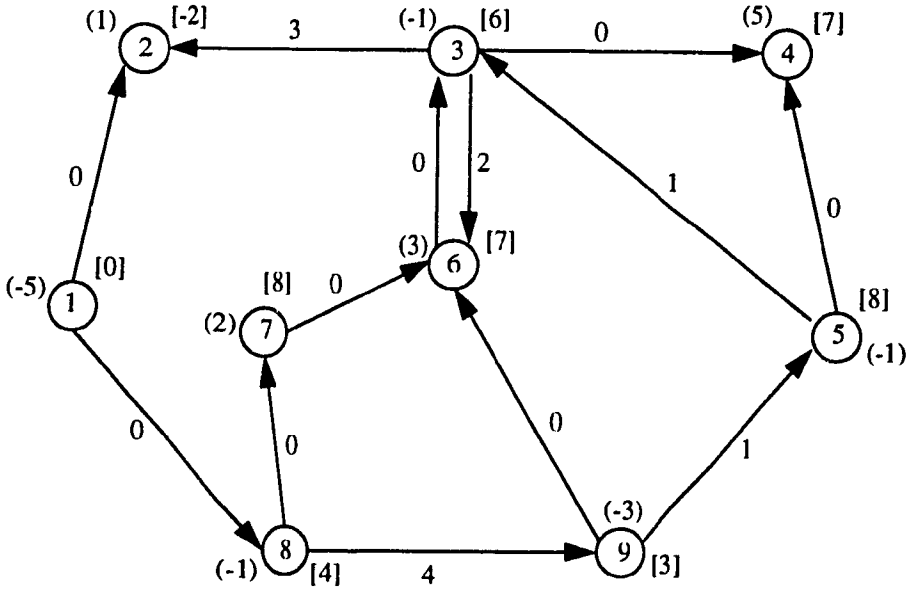


Figure 5.9 The optimum solution of the graph G.

Formal Presentation of the Concurrent Pivot Algorithm

```
procedure initialize_cpivots ( i )
    clusterWeight [i] ← nodeWeight [i];
    source [i] ← i;
    num_tree_edges [i] ← new_num_tree_edges [i] ← | treeNodes [i] |;
    if (num_tree_edges [i] = 1)
    then notFinished ← TRUE
    end if;
    cluster_firing_no [i] ← INFINITY;
    positive_fire_flag ← negative_fire_flag ← FALSE;
end procedure {initialize_cpivots}
procedure positive_fire_check ( i )
    node ← the (only) member in treeNodes [i];
    if (node ∉ inNodes [i])
    then positive_fire_flag ← TRUE;
    end if;
end procedure {positive_fire_check}
procedure check_if_branches_in_right_direction ( i )
    if (num_tree_edges [i] = 1)
    then
        if (clusterWeight [i] > 0)
        then positive_fire_check ( i )
        elsif (clusterWeight [i] < 0)
```

```
            then negative_fire_check ( i )
            end if
        end if
    end procedure {check_if_branches_in_right_direction}
    procedure get_min_positive_firing_no ( i )
        fno ← INFINITY;
        for all j in inNodes [i] do
            if (token [j] [i] < fno and source [j] ≠ source [i])
            then fno ← token [j] [i]
            end if;
        end for;
        return fno;
    end procedure {get_min_positive_firing_no}
    procedure calculate_positive_firing_no ( i )
        if (num_tree_edges [i] < 2 and clusterWeight [i] > 0)
        then
            fno ← get_min_positive_firing_no ( i );
            lock if (fno < cluster_firing_no [source [i]])
                then cluster_firing_no [source [i]] ← fno
                end if;
        end if
    end procedure {calculate_positive_firing_no}
```

```
procedure fire (i, fno)
    if (fno = INFINITY)
    then unbounded ← TRUE
    else
        firing_no [i] ← firing_no [i] + fno;
        for all j in inNodes [i] do
            token [j] [i] ← token [j] [i] - fno
        end for;
        for all j in outNodes [i] do
            token [i] [j] ← token [i] [j] + fno
        end for;
    end if
end procedure {fire}
procedure positive_fire ( i )
    if (num_tree_edges [i] < 2 and clusterWeight [i] > 0)
    then
        fno ← cluster_firing_no [source [i]];
        if (fno > 0)
        then notOptimal ← TRUE
        end if;
        fire (i, fno);
    end if
end procedure {positive_fire}
```

```
procedure get_min_negative_firing_no ( i )
    fno ← INFINITY;
    for all j in outNodes [i] do
        if (token [i] [j] < fno and source [j] ≠ source [i])
        then fno ← token [i] [j]
        end if
    end for;
    return fno;
end procedure {get_min_negative_firing_no}
procedure calculate_negative_firing_no ( i )
    if (num_tree_edges [i] < 2 and clusterWeight [i] < 0)
    then
        fno ← get_min_negative_firing_no ( i );
        lock if (fno < cluster_firing_no [source [i]])
            then cluster_firing_no [source [i]] ← fno
            end if;
    end if
end procedure {calculate_negative_firing_no}
procedure negative_fire ( i )
    if (num_tree_edges [i] < 2 and clusterWeight [i] < 0)
    then
        fno ← cluster_firing_no [source [i]];
        if (fno > 0)
```

```
        then notOptimal ← TRUE
        end if;
        fire (i, - fno);
    end if
end procedure { negative_fire }
procedure climb_up_tree ( i )
    if (num_tree_edges [i] = 1)
    then
        new_num_tree_edges [i] ← 0;
        if (treeNodes [i] ≠ ∅)
        then
            node ← the (only) member in treeNodes [i];
            source [i] ← node;
            lock clusterWeight [node] ← clusterWeight [node] + clusterWeight [i];
            lock new_num_tree_edges[node] ← new_num_tree_edges[node] - 1;
            lock delete i from treeNodes [node];
        end if;
    end if
end procedure {climb_up_tree}
procedure find_new_sources ( i )
    num_tree_edges [i] ← new_num_tree_edges [i];
    if (num_tree_edges [i] = 1)
    then notFinished ← TRUE;
```

```
    elsif (num_tree_edges [i] = 0)
    then
        s ← i;
        while (s ≠ source [s]) do
            s ← source [s]
        end while;
        source [i] ← s;
        clusterWeight [i] ← clusterWeight [s];
    end if;
end procedure { find_new_sources }
procedure concurrent_pivots ()
{ main algorithm }
    notFinished ← FALSE;
    for i ← 0 to N - 1 do in parallel
        initialize_cpivots ( i )
    end for;
    while (notFinished) do
        notFinished ← FALSE;
        for i ← 0 to N - 1 do in parallel
            check_if_branches_in_right_direction ( i )
        end for;
        if (positive_fire_flag)
        then
```

```
    positive_fire_flag ← FALSE;
    for i ← 0 to N - 1 do in parallel
        calculate_positive_firing_no ( i )
    end for;
    for i ← 0 to N - 1 do in parallel
        positive_fire ( i )
    end for;
    if (unbounded)
    then exit_with_error_message ()
    end if;
end if;
if (negative_fire_flag)
then
    negative_fire_flag ← FALSE;
    for i ← 0 to N - 1 do in parallel
        calculate_negative_firing_no ( i )
    end for;
    for i ← 0 to N - 1 do in parallel
        negative_fire ( i )
    end for;
    if (unbounded)
    then exit_with_error_message ()
    end if;
```

```
        end if;
        for i ← 0 to N - 1 do in parallel
            climb_up_tree ( i )
        end for;
        for i ← 0 to N - 1 do in parallel
            find_new_sources ( i )
        end for;
    end while;
end procedure { concurrent_pivots }
```

5.4.6 Avoidance of Cycling

We have incorporated in the Concurrent pivoting phase, Bland's anti-cycling rule [39] to avoid occurrence of cycling. Bland's anti-cycling rule states that if there is a choice in picking the entering and leaving variables, then always pick the candidate x_k that has the smallest subscript k.

Cycling can occur in our algorithm only if there are more than one degenerate pivot in one iteration in the concurrent pivots phase. The degenerate pivots can be easily recognized because the firing number in this case is zero. Also one can number the edges in such a way that the smallest edge can be recognized by looking at the node numbers. This can be done as follows.

1 All incoming edges at node i will have lower number than any incoming edge at node j if $i < j$.

2 Of any incoming edges at node i, edge (k, i) will have lower number than the edge (l, i) if $k < l$.

The Bland's anti-cycling rule now can be easily incorporated in our algorithm by simply recognizing degenerate pivots and allowing the least numbered node to leave and another least numbered node to enter the spanning tree. In the case of negative firing, we will compare the nodes at the other end of the edges that are entering/leaving the tree.

The formal presentation of main algorithm is given next.

Parallel Network Dual Simplex Algorithm

```
procedure MNDS ()
{ main algorithm }
    feasible ();
    do
        notOptimal ← FALSE;
        do
            no_of_nodes ← 0;
            clustering ();
            if (no_of_nodes > 1)
            then
                contraction ();
                shortest_path ();
                firing ();
            end if;
        while (no_of_nodes > 1);
        build_tree ();
        concurrent_pivots ();
    while (notOptimal);
end procedure { MNDS }
```

5.5 Summary

In this chapter, we have presented a new approach to solve the dual transshipment problem. This approach, called the Modified Network Dual Simplex (MNDS), involves repeated applications of three basic algorithms: Algorithm FEASIBLE to test feasibility of the DTP, Algorithm SHORTEST-PATH to compute shortest paths in a multiple source - multiple sink network and Concurrent pivots. Starting with a feasible solution, the nodes attempt to improve the value of the objective as much as possible using node and cluster firing operations. When no more improvement is possible, we will have reached a basic feasible solution at which point the algorithm improves the objective through concurrent pivot operations. This phase - node/cluster firing and concurrent pivots - is repeated until we reach a basic feasible solution which does not permit pivot operations. Unlike the traditional network dual simplex method, MNDS does not move from one basic feasible solution to another. Implementing node/cluster firings through shortest path computations and carrying out concurrent pivots efficiently and without destroying feasibility are the distinct features of MNDS. An experimental evaluation of MNDS will be discussed in Chapter 7.

CHAPTER 6

A CLUSTER-BASED PARALLEL ALGORITHM FOR THE DUAL TRANSSHIPMENT PROBLEM

In this chapter, we develop a novel approach to the solution of the dual transshipment problem. Towards this end, we first establish a new characterisation of the optimum solutions of DTP. This characterisation relates the structure of an optimum solution to the structure of certain zero-residual token clusters rooted at the negative weighted nodes. Our approach, to be called the Cluster-Based Dual Simplex (CBDS) method, uses concepts and results from the theory of marked graphs. CBDS has several distinguishing features which make it amenable for an efficient parallel implementation.

The CBDS method to the solution of the DTP involves repeated applications of three basic steps, namely, feasibility testing, cluster optimization and performing concurrent pivots. In the cluster optimization step, certain clusters rooted at negative-weighted nodes are constructed using an algorithm which resembles Prim's approach to the minimum cost spanning tree problem [135]. This step is the most distinguishing feature of CBDS. In the following sections, we develop CBDS and discuss essential features of its implementation in a shared memory programming environment.

In this chapter, unless otherwise stated, firing would mean negative firing.

6.1 Basic Definitions

Consider a directed graph G = (V, E). Let A denote the incidence matrix of G, M denote the column vector of edge tokens and W denote the colum vector of node weights. Recall again that the dual transshipment problem (Section 2.2) is a linear program defined as follows:

Maximize: $W^t Y$

subject to

$$A^t Y \geq -M^t \tag{6.1}$$

$$Y \geq 0, \tag{6.2}$$

where Y is the column vector of node firing numbers.

Clearly, in DTP, we seek to determine Y such that $W^t Y$ is maximum and (6.1) and (6.2) are satisfied. Any vector Y which satisfies (6.1) and (6.2) is a feasible solution of the DTP.

Given a feasible solution $y_1, y_2, ..., y_n$ of the DTP, the residual token on the edge (i, j) is given by

$$y_i - y_j + m_{ij}.$$

Note that feasible solution Y guarantees that the residual tokens are all non-negative. We can easily show that the residual edge tokens define a feasible Y to within an additive constant [44], [162]. So, we can view the residual edge tokens as defining a solution to the 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* if

i) if a cluster $C_i \in S_k$, then all clusters that are linked to C_i are also in S_k, and

ii) if a cluster $C_j \notin S_k$, then $C_i \cap C_j = \emptyset$, for all $C_i \in S_k$.

6.2 A Characterisation of Optimum Solutions for the DTP

Recall (Section 2.4) that a basic feasible solution Y of a DTP is optimum iff no dual pivot operation is permissible under this solution. This is, in fact, the simplex criterion for optimality. We now present an alternate characterisation of optimality in terms of the clusters defined in the previous section.

Theorem 6.1

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 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 $v_1, v_2, ..., v_k$ be the negative nodes in S and $C_1, C_2, ..., C_k$ 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, $C_i - S \neq \emptyset$, for some C_i, $1 \leq i \leq k$, 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. ❑

6.3 Outline of a Cluster-Based Algorithm for the DTP

The optimality criteria proved in Theorem 6.1 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 $k \leq 3$. So, our approach 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 concurrent pivots as described in the previous chapter (Step 6).

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.

An outline of the above algorithm is given in Figure 6.1.

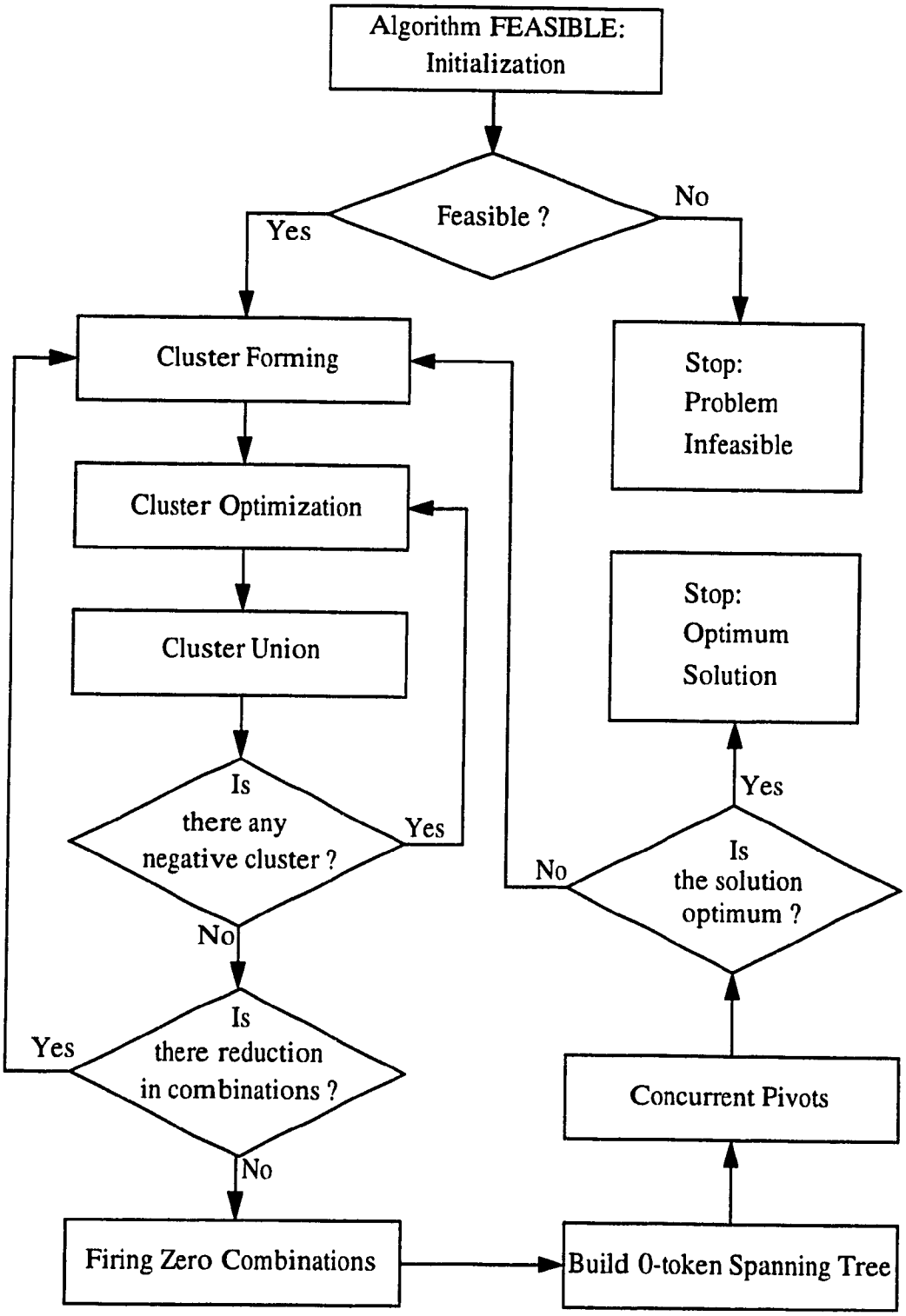
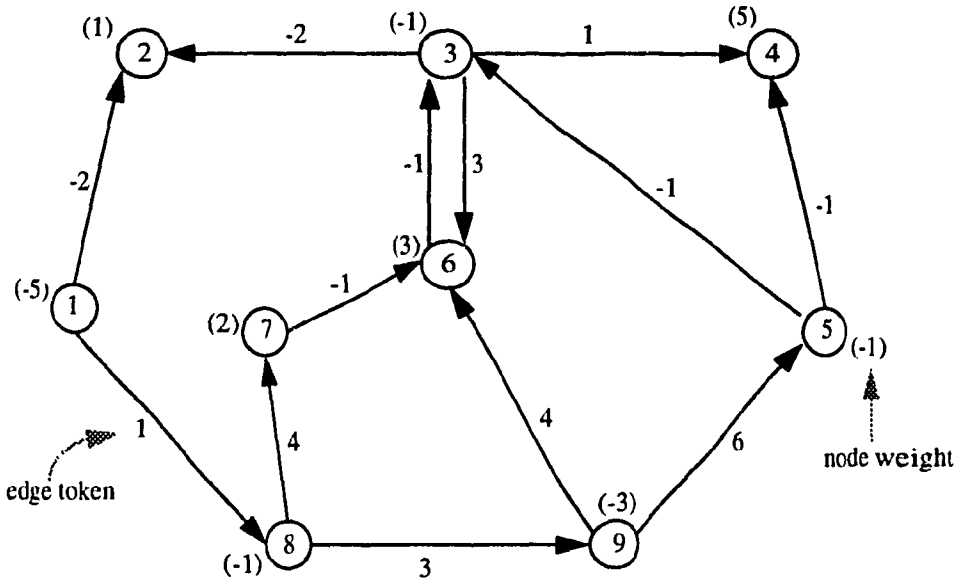


Figure 6.1 Cluster-Based Dual Simplex Method for the DTP

The graph G shown in Figure 6.2 will be used to illustrate each step of our algorithm. Again this is the same example we used to illustrate our other algorithms in the previous chapters.

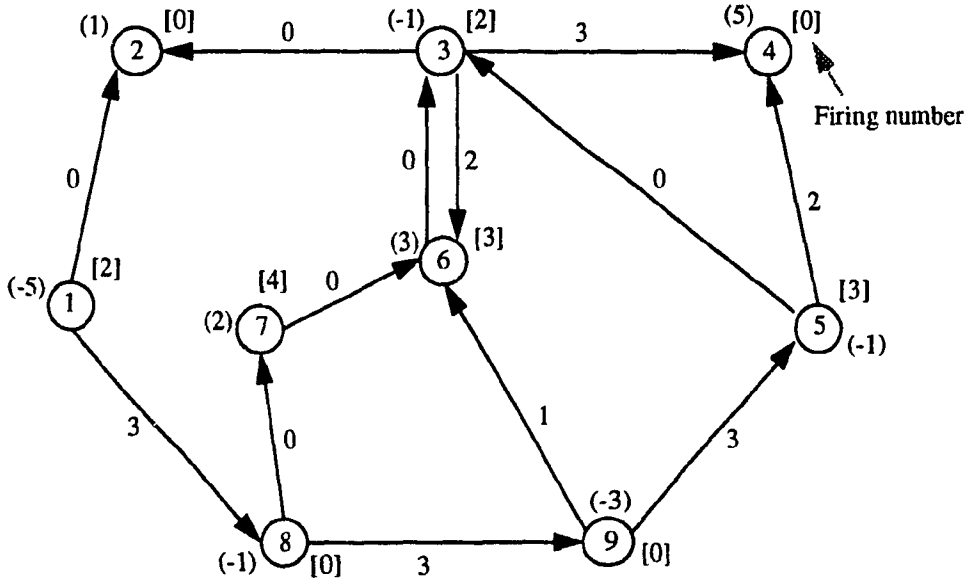


Node numbers are shown inside nodes.
Node Weights and edge tokens are as shown.

Figure 6.2 An example to illustrate our algorithm: Given graph G.

6.4 Feasibility Testing

We apply Algorithm FEASIBLE to test the feasibility of the problem as explained in the previous chapters. If the problem is not feasible, then the algorithm stops here; otherwise it will proceed to the next step. Figure 6.3 shows our graph after the application of Algorithm FEASIBLE.



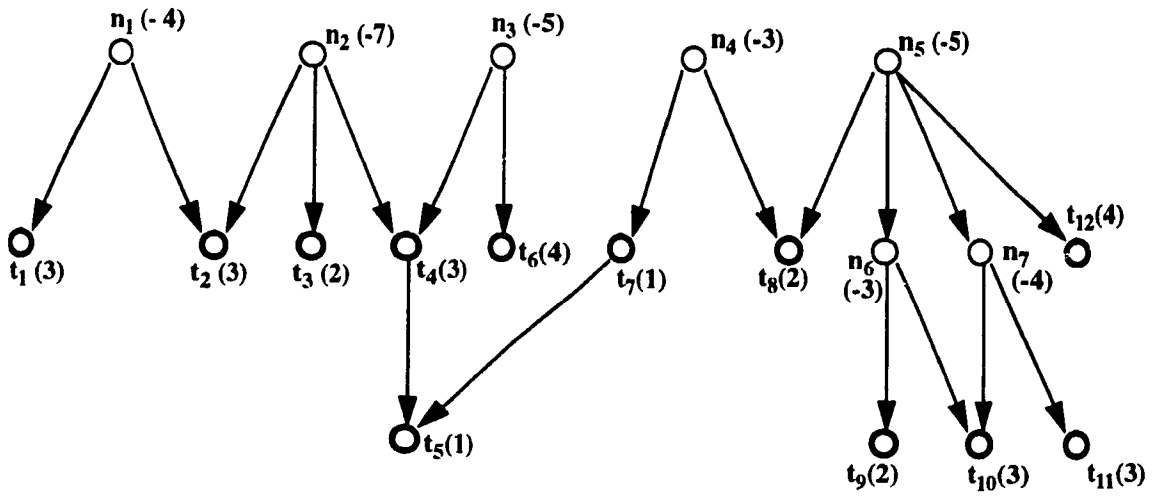
Firing numbers of nodes are shown in square brackets.

Figure 6.3 The graph G after feasibility testing.

6.5 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. Different examples of overlapping clusters are shown in the Figure 6.4.



Circles represented by n_i are negative nodes and dark lined circles represented by t_i are 0-token subtrees consisting of only non-negative nodes. The weights are given in parentheses.

Figure 6.4 An illustration of different overlapping clusters.

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 implementation details of these phases.

n : an integer variable. It denotes the number of nodes in the graph G.

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.

outNodes : an array of sets. *outNodes [i]* is a set containing all the outnodes at node *i*.

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.

zeroNodes : an array of sets. *zeroNodes [i]* contains all zero outnodes at node *i*.

isSimpleClusters : a boolean variable. *isSimpleClusters* equal to TRUE indicates that the clusters formed are simple clusters at each negative node; otherwise, it indicates that the clusters are combinations..

6.5.1 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]*.

A formal presentation of Cluster Initialization is given next.

```
procedure cluster_initialization (i)
    zeroNodes [i] ← ∅;
    sources [i] ← newSources [i] ← ∅;
    if (nodeWeight [i] < 0)
    then
        add i to sources [i];
        clusterWeight [i] ← nodeWeight [i];
        clusterFiringNo [i] ← INFINITY;
    else clusterWeight [i] ← 0;
    end if;
end procedure { cluster_initialization }
```

6.5.2 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. (See procedures get_new_zeroNodes, propagate_sources_to_new_zeroNodes and calculate_zeroNodes.)

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. (See procedures prop-

agate_new_sources and calculate_cluster_weight.)

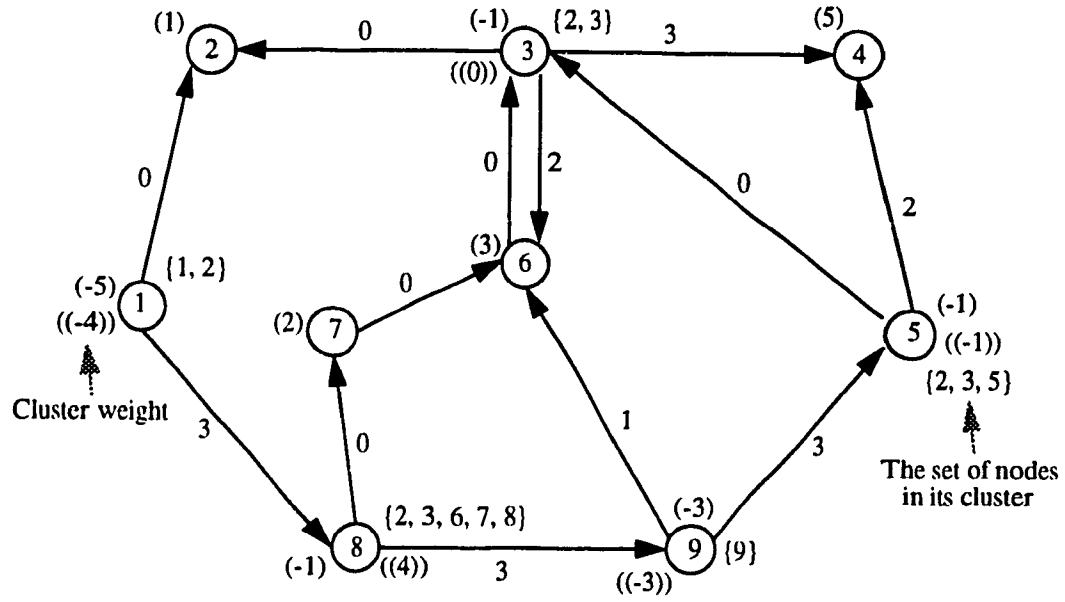


Figure 6.5 The graph G after clustering.

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. The cluster for each negative node in our example is shown in braces in Figure 6.5. The weight of each cluster is also shown in double parenthesis at each negative node.

A formal presentation of the cluster expansion phase is given next.

```
procedure get_new_zeroNodes (i)
{ Find all zero outnodes at node i }
    newZnodes ← ∅;
    for all j in outNodes [i] do
        temp ← token [i] [j] + firingNo [i] - firingNo [j];
        if (temp = 0)
```

```
        then add j to newZnodes;
        end if;
    end for;
    return newZnodes;
end procedure { get_new_zeroNodes }
procedure propagate_sources_to_new_zeroNodes (i, newZnodes)
{ Propagate sources information to all new zero outnodes }
    if (sources [i] ≠ ∅)
    then
        znodes ← newZnodes - zeroNodes [i];
        if (znodes ≠ ∅)
        then
            for all j in znodes do
                lock newSources [j] ← newSources [j] ∪ sources [i]
            end for;
            notFinished ← TRUE;
        end if;
    end if;
end procedure { propagate_sources_to_new_zeroNodes }
procedure calculate_zeroNodes ( i )
    newZnodes ← get_new_zeroNodes ( i );
    propagate_sources_to_new_zeroNodes (i, newZnodes);
    zeroNodes [i] ← newZnodes;
```

```
        if (nodeWeight [i] < 0)
        then clusterFiringNo [i] ← INFINITY;
        end if;
    end procedure { calculate_zeroNodes }
    procedure propagate_new_sources ( i )
        temp ← newSources [i] - sources [i];
        if (temp ≠ ∅)
        then
            sources [i] ← sources [i] ∪ temp;
            if (zeroNodes [i] ≠ ∅)
            then
                for all j in zeroNodes [i] do
                    lock newSources [j] ← newSources [j] ∪ temp
                end for;
                notFinished ← TRUE;
            end if;
            for all k in temp do
                lock clusterWeight [k] ← clusterWeight [k] + nodeWeight [i]
            end for;
        end if;
    end procedure { propagate_new_sources }
    procedure cluster_expansion ()
        notFinished ← FALSE;
```

```
    for i ← 0 to n - 1 do in parallel
        calculate_zeroNodes ( i )
    end for;
    while (notFinished) do
        notFinished ← FALSE;
        for i ← 0 to n - 1 do in parallel
            propagate_new_sources ( i )
        end for;
    end while;
end procedure { cluster_expansion }
```

Now, we will put together the above two phases and give the main procedure for the Cluster Forming step.

```
procedure cluster_forming ()
{ main procedure }
    isSimpleClusters ← TRUE;
    for i ← 0 to n - 1 do in parallel
        form_initial_clusters ( i )
    end for;
    expand_clusters ();
end procedure { cluster_forming }
```

6.6 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.

6.6.1 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. (See procedure fire.)

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.

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 6.5.2.

For our example, the graph at the end of the first iteration of cluster firing will be as shown in Figure 6.6. At this point, all the clusters are non-negative weighted and the algorithm moves to the next step.

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

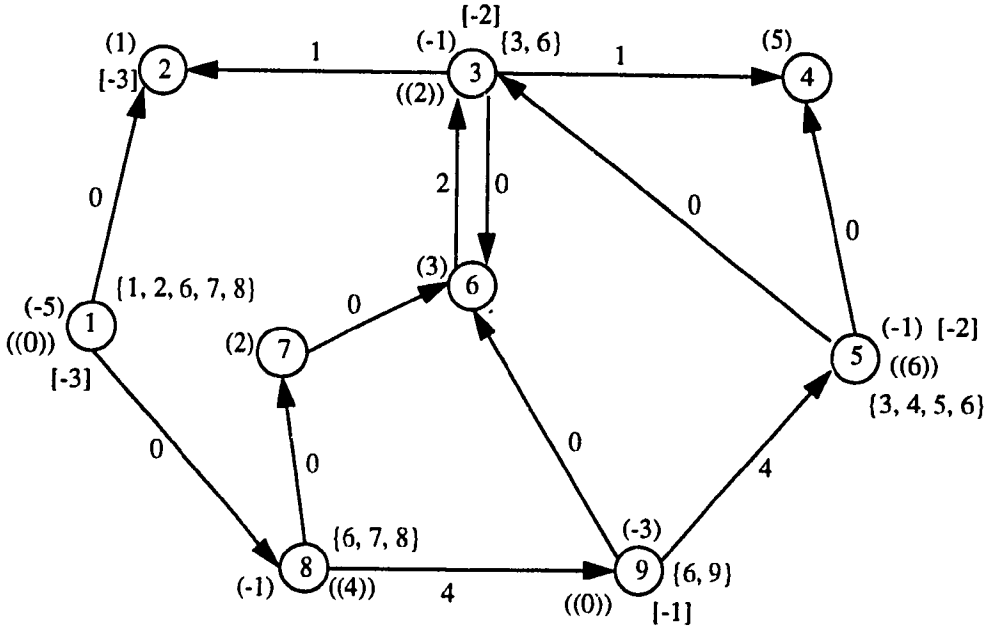


Figure 6.6 The graph G with new clusters after cluster firing.

A formal presentation of the cluster optimization step is given next.

```
procedure is_any_negative_cluster (i)
    if (nodeWeight [i] < 0)
    then
        if (clusterWeight [i] < 0)
        then notFinished ← TRUE
        end if
    else
        temp ← ∅;
        for all j in sources [i] do
            temp ← temp ∪ sources [j]
        end for;
```

```
        sources [i] ← temp;
    end if
end procedure { is_any_negative_cluster }
procedure find_cluster_firingNo (i, temp_outnodes, temp_sources)
    while (temp_sources ≠ ∅ and temp_outnodes ≠ ∅) do
        fno ← INFINITY;
        for all j in temp_outnodes do
            temp ← token [i] [j] + firingNo [i] - firingNo [j];
            if (temp < fno)
            then
                fno ← temp;
                min_outnode ← j;
            end if;
        end for;
        common_sources ← temp_sources ∩ sources [min_outnode];
        other_sources ← temp_sources - common_sources;
        oRoot ← first member in sources [min_outnode];
        for all k in other_sources do
            lock clusterFiringNo [k];
                if (fno < clusterFiringNo [k])
                then
                    clusterFiringNo [k] ← fno;
                    otherRoot [k] ← oRoot;
```

```
            end if:
            unlock clusterFiringNo [k];
        end for;
        temp_sources ← common_sources;
        delete min_outnode from temp_outnodes;
    end while;
end procedure { find_cluster_firingNo }
procedure calculate_cluster_firingNo ( i )
    temp_outnodes ← outNodes [i] - zeroNodes [i];
    temp_sources ← sources [i];
    find_cluster_firingNo (i, temp_outnodes, temp_sources);
end procedure { calculate_cluster_firingNo }
procedure to_fire_or_not ( i )
{ This is done by sources only. They will decide for their cluster whether to
participate in the firing pulse or not }
    if (nodeWeight [i] < 0)
    then
        if (clusterWeight [i] ≥ 0)
        then fno ← 0
        else fno ← clusterFiringNo [i]
        end if;
        for all j in sources [i] do
            if (clusterWeight [j] < 0 and clusterFiringNo [j] > fno)
```

```
            then fno ← clusterFiringNo [j]
            end if
        end for;
        clusterFiringNo [i] ← fno;
    end if
end procedure { to_fire_or_not }
procedure fire ( i )
    fno ← 0;
    new_sources ← ∅;
    for all j in sources [i] do
        if (clusterFiringNo [j] > fno)
        then
            fno ← clusterFiringNo [j];
            new_sources ← ∅;
            add j to new_sources;
        elsif (clusterFiringNo [j] = fno)
        then add j to new_sources
        end if
    end for;
    if (fno = INFINITY)
    then unbounded ← TRUE
    elsif (fno > 0)
    then
```

```
        rm_sources ← sources [i] - new_sources;
        if (rm_sources ≠ ∅)
        then
            isReduced ← TRUE;
            for all k in rm_sources do
                lock clusterWeight [k] ← clusterWeight [k] - nodeWeight [i]
            end for;
            sources [i] ← sources [i] - rm_sources;
            newSources [i] ← newSources [i] - rm_sources;
            firingNo [i] ← firingNo [i] - fno;
        end if;
    end if;
end procedure { fire }
procedure cluster_optimization ()
{ main procedure }
    noIterations ← 0;
    do
        notFinished ← FALSE;
        noIterations ← noIterations + 1;
        for i ← 0 to n - 1 do in parallel
            is_any_negative_cluster ( i )
        end for;
        if (notFinished)
```

```
        then
            for i ← 0 to n - 1 do in parallel
                calculate_cluster_firingNo ( i )
            end for;
            if (isSimpleClusters)
            then
                for i ← 0 to n - 1 do in parallel
                    to_fire_or_not ( i )
                end for
            end if;
            for i ← 0 to n - 1 do in parallel
                fire ( i )
            end for;
            if (not unbounded)
            then cluster_expansion ()
            end if;
        end if;
        if (noIterations ≥ n)
        then unbounded ← TRUE
        end if;
    while (notFinished and not unbounded);
    if (unbounded)
    then
```

```
        print "The problem is unbounded";
        stop;
    end if;
end procedure { cluster_optimization }
```

6.7 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 6.1.

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 the variable *combination* to propagate this root information. 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.

The following additional data structure is used in this step.

combination : an array of sets. The variable *combination [i]* contains the set of the sources that should be combined with the source *i*.

A formal presentation of the Cluster Union step is given next.

```
procedure propagate_min_source ( i )
    if (nodeWeight [i] < 0)
    then
        clusterWeight [i] ← 0;
        s ← first member in sources [i];
        for all j in sources [i] do
            add s to combination [j]
        end for;
    elsif (zeroNodes [i] = ∅ and sources [i] ≠ ∅)
    then
        s ← first member in sources [i];
        for all j in sources [i] do
            add s to combination [j]
```

```
        end for;
    end if
end procedure { propagate_min_source }
procedure get_last_link (s)
    t ← first member in combination [s];
    while (t ≠ s) do
        s ← t;
        t ← first member in combination [s];
    end while;
    return s;
end procedure { get_last_link }
procedure find_combination_root ( i )
    if (nodeWeight [i] < 0 and combination [i] ≠ ∅)
    then
        flag ← FALSE;
        t ← first member in combination [i];
        s ← get_last_link (t);
        for all j in combination [i] do
            t ← get_last_link (j);
            delete j from combination [i];
            if (t < s)
            then
                add t to combination [s];
```

```
                s ← t;
                flag ← TRUE;
            elsif (s < t)
            then
                add s to combination [t];
                flag ← TRUE;
            end if;
        end for;
        add s to combination [i];
        if (flag)
        then notFinished ← TRUE
        end if;
    end if
end procedure { find_combination_root }
procedure find_combination_weight ( i )
    if (sources [i] ≠ ∅)
    then
        j ← first member in sources [i];
        if (combination [i] ≠ ∅)
        then s ← first member in combination [j]
        else s ← j
        end if;
        sources [i] ← ∅;
```

```
            add s to sources [i];
            lock clusterWeight [s] ← clusterWeight [s] + nodeWeight [i]
        end if
    end procedure { find_combination_weight }
    procedure is_there_any_negative_combination ( i )
        if (nodeWeight [i] < 0)
        then
            combination [i] ← ∅;
            if (clusterWeight [i] < 0)
            then notCompleted ← TRUE
            else lock no_of_clusters ← no_of_clusters + 1
            end if;
        elsif (nodeWeight [i] = 0)
        lock no_of_clusters ← no_of_clusters + 1
        end if
    end procedure { is_there_any_negative_combination }
    procedure cluster_union ()
        for i ← 0 to n - 1 do in parallel
            propagate_min_source ( i )
        end for;
        do
            notFinished ← FALSE;
            for i ← 0 to n - 1 do in parallel
```

```
            find_combination_root ( i )
        end for;
    while (notFinished);
    for i ← 0 to n - 1 do in parallel
        find_combination_weight ( i )
    end for;
    for i ← 0 to n - 1 do in parallel
        is_there_any_negative_combination ( i )
    end for;
    if (isSimpleClusters)
    then isSimpleClusters ← FALSE
    end if;
end procedure { cluster_union }
```

6.8 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 fir-

ing 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 as explained before. This is decided on a first-come first-serve basis using a set variable *firingGroup*,

i) If *firingGroup* = ∅, then cluster *i* will add *i* and *j* to *firingGroup* and set *isFire [i]* = TRUE, or

ii) If *firingGroup* ≠ ∅ and $i \notin$ *firingGroup* and $j \notin$ *firingGroup*, then cluster *i* will add *i* and *j* to *firingGroup* and set *isFire [i]* = TRUE.

Otherwise, it will set *isFire [i]* = FALSE.

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.

The following additional data structures are used in this step.

firingGroup : a set variable. This variable will be used to decide which zero combination will be allowed to fire.

isRoot : an array of boolean. The variable *isRoot [i]* is TRUE if node *i* is the root of the combination in which node *i* is present.

isFire : an array of boolean. The variable *isFire [i]* is TRUE if node *i* is the root of the combination and if that combina-

tion is allowed to fire.

otherRoot : an array of integers. The variable *otherRoot [i]* contains the root of the other combination into which it is going to merge, if node *i* is the root of the combination and if that combination is allowed to fire.

The formal presentation of this step is given next.

```
procedure calculate_zeroCluster_firingNo ( i )
    temp_outnodes ← outNodes [i] - zeroNodes [i];
    if (temp_outnodes ≠ ∅)
    then
        temp_sources ← sources [i];
        if (temp_sources = ∅ and nodeWeight [i] = 0)
        then
            add i to temp_sources;
            add i to sources [i];
        end if;
        find_cluster_firingNo (i, temp_outnodes, temp_sources);
    end if
end procedure { calculate_zeroCluster_firingNo }
procedure can_I_fire ( i )
    isFire [i] ← FALSE;
    j ← first member in sources [i];
    if (i = j)
    then
        isRoot [i] ← TRUE;
```

```
        if (clusterFiringNo [i] < INFINITY)
        then
            my_group ← ∅;
            add i to my_group;
            add otherRoot [i] to my_group;
            lock firingGroup;
                if (my_group ∩ firingGroup = ∅)
                then
                    isFire [i] ← TRUE;
                    firingGroup ← firingGroup ∪ my_group;
                end if;
            unlock firingGroup;
        end if;
    else isRoot [i] ← FALSE
    end if;
end procedure { can_I_fire }
procedure fire_zero_combination ( i )
    if (sources [i] ≠ ∅)
    then
        if (isRoot [i])
        then j ← i
        else j ← first member in sources [i]
        end if;
```

```
        if (isFire [j])
        then
            fno ← clusterFiringNo [j];
            firingNo [i] ← firingNo [i] - fno;
            if (i = j)
            then
                if (i < otherRoot [i])
                then
                    lock sources [otherRoot [i]];
                        sources [otherRoot [i]] ← ∅;
                        add i to sources [otherRoot [i]];
                    unlock sources [otherRoot [i]];
                else
                    lock sources [i];
                        sources [i] ← ∅;
                        add otherRoot [i] to sources [i];
                    unlock sources [i];
                end if
            end if;
        end if;
    end if
end procedure { fire_zero_combination }
procedure merge_zero_combinations ( i )
```

```
        zeroNodes [i] ← get_new_zeroNodes (i);
        if (sources [i] ≠ ∅)
        then
            j ← first member in sources [i];
            if (i ≠ j)
            then
                k ← first member in sources [j];
                lock sources [i];
                    sources [i] ← ∅;
                    add k to sources [i];
                unlock sources [i];
                if (isRoot [i])
                then lock clusterWeight [k] ← clusterWeight [k] + clusterWeight [i];
                end if;
            else
                no_of_clusters ← no_of_clusters + 1;
                clusterFiringNo [i] ← INFINITY;
            end if;
        end if
    end procedure { merge_zero_combinations }
    procedure zero_combinations ()
        while (no_of_clusters > 1) do
            firingGroup ← ∅;
```

```
        for i ← 0 to n - 1 do in parallel
            calculate_zeroCluster_firingNo ( i )
        end for;
        for i ← 0 to n - 1 do in parallel
            can_!_fire ( i );
        end for;
        for i ← 0 to n - 1 do in parallel
            fire_zero_combination ( i )
        end for;
        no_of_clusters ← 0;
        for i ← 0 to n - 1 do in parallel
            merge_zero_combinations ( i )
        end for;
    end while
end procedure { zero_combinations }
```

6.9 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 is explained in detail, in the previous chapter. So, we will not elaborate it further. If the solution is not optimum, then the algorithm goes back to the Cluster Forming step.

Now, for our example, the clusters in Figure 6.6 can be redrawn to bring out the different clusters as shown in Figure 6.7. This will also be a 0-token spanning tree to be used in the concurrent pivots step. Since it is not optimum, there will be some firings in

concurrent pivots step. The new clusters are as shown in Figure 6.8. Since the solution is optimum at this stage, the algorithm terminates at this point.

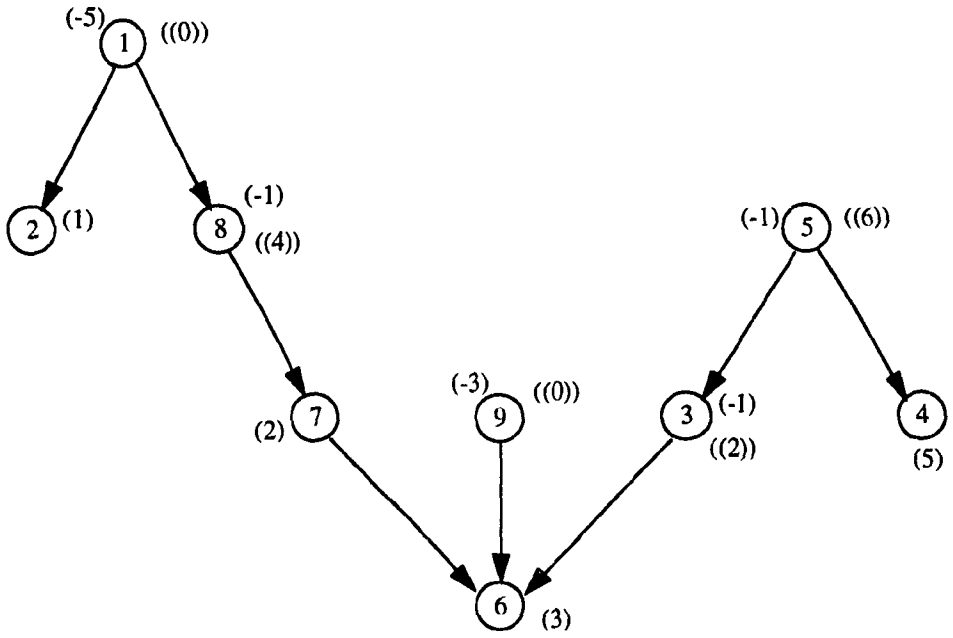


Figure 6.7 The clusters of the graph in Figure 6.6.

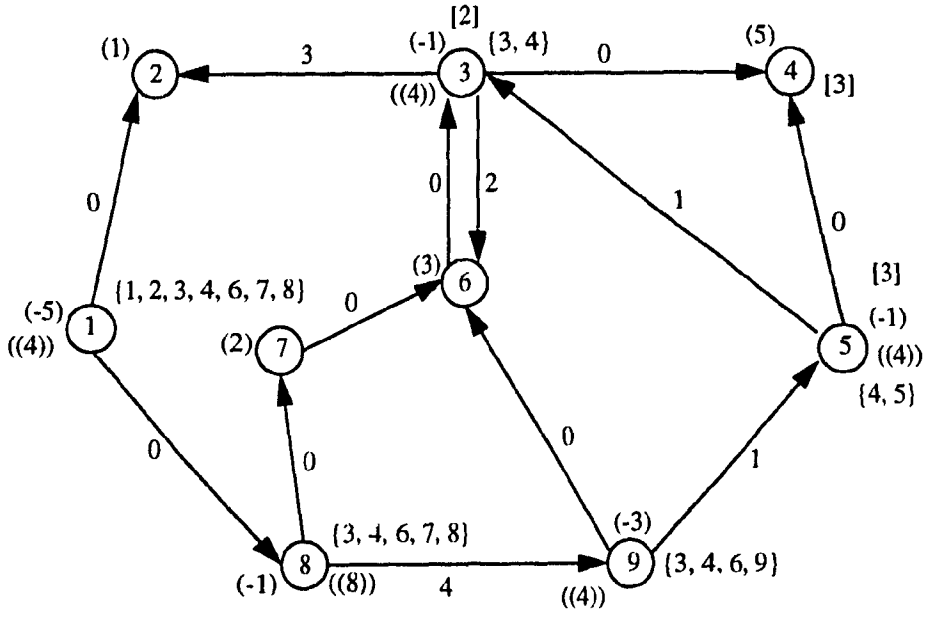


Figure 6.8 The graph G with new clusters after concurrent pivots.

6.10 Summary of the Parallel Algorithm

A formal presentation of our parallel algorithm for the DTP, involving procedures described in the previous sections is given next.

Parallel Cluster-Based Dual Simplex Algorithm

```
procedure CBDS ()
{ main procedure }
    if (not feasible ())
    then STOP
    end if;
    unbounded ← FALSE;
    do
       do
          form_clusters ();
          isReduced ← FALSE;
          do
             cluster_optimization ();
             if (not isReduced)
             then
                notCompleted ← FALSE;
                no_of_clusters ← 0;
                cluster_union ();
             end if;
          while (notCompleted and not isReduced);
       while (isReduced);
       zero_combinations ()
```

```
        notOptimal ← FALSE;
        concurrent_pivots ();
    while (notOptimal);
end procedure { CBDS }
```

6.11 Summary

In this chapter, we have presented a novel approach to solve the dual transshipment problem. This approach, called the Cluster-Based Dual Simplex (CBDS), involves repeated applications of three basic algorithms: Algorithm FEASIBLE to test feasibility of the DTP, the Cluster Optimization algorithm and Concurrent Pivots. Starting with a feasible solution, the negative weighted nodes attempt to improve the value of the objective as much as possible using cluster firing operations. When no further improvement is possible, the algorithm will have constructed non-negative weighted clusters rooted at the negative nodes. The cluster optimization step is carried out using an algorithm similar to Prim's min-cost spanning tree algorithm. Repeated applications of cluster optimization will result in a basic feasible solution. The algorithm then attempts further improvement of the objective through concurrent pivot operations. This phase - cluster optimization and concurrent pivots - is repeated until we reach a basic feasible solution which does not permit any pivot operations. Like the MNDS method of the previous chapter, CBDS does not move from one basic solution to another (in contrast to the traditional network dual simplex method). The cluster optimization step distinguishes CBDS from MNDS. An experimental evaluation of CBDS will be discussed in Chapter 7.

CHAPTER 7

EXPERIMENTAL EVALUATION

In this chapter, we report results of an experimental evaluation of the three parallel algorithms - Network Primal Dual, MNDS and CBDS - developed in Chapters 4 - 6. We have implemented these algorithms on the BBN Butterfly parallel computer (Mach 1000). We also present certain conclusions on the relative merits of these algorithms.

For our experimentation, we have used NETGEN, a program for generating a variety of network problems. NETGEN can generate capacitated and uncapacitated transshipment problems as well as assignment problems. This program was originally developed by a group of researchers at the University of Texas at Austin. A detailed discussion of the development and implementation of NETGEN may be found in [84].

We present two sets of results. In Tables 7.1 - 7.5, we present results of implementations of the network primal-dual, MNDS and CBDS algorithms. These results are presented for graphs of varying sizes generated by NETGEN. In each case, we have presented results for up to 10 processors. In Tables 7.6 - 7.10, we present similar results for the MNDS and CBDS algorithms.

Following are certain general conclusions based on the tabulated results. An iteration in the case of the network primal-dual method will refer to one application of the shortest path and the max-flow algorithms. For the MNDS algorithm, this refers to the step in which a basic feasible solution is generated using shortest path and clustering algorithms. For the CBDS algorithm, an iteration refers to the step in which a basic feasible solution is generated using cluster optimization and clustering algorithms.

An examination of the tabulated results shows that the network primal dual method is considerably slower than the other two algorithms. We have observed that the max-flow algorithm used in the network primal dual method performs an excessively large

number of relabeling operations before performing a push operation. Since a large number of iterations are required in the network primal dual case, these relabeling operations seem to be the main source of the poor performance of the network primal dual method. We expect better performance if the Goldberg-Tarjan max-flow algorithm is replaced by Dinic's and MPM algorithms [50], [102].

Tables 7.1 - 7.10 indicate that the CBDS algorithm is, in general, faster than the MNDS algorithm. Also, CBDS requires lesser number of iterations than MNDS. This confirms our expectation that in each iteration of the CBDS, considerable amount of improvement in the objective value is achieved than in each iteration of MNDS. Also, CBDS, in general, gives better speed up than MNDS. The improved speed up observed for CBDS is also in agreement with our expectation. In fact, CBDS was designed specifically to achieve better speed up than MNDS. In CBDS, almost all nodes remain active during a good part of each iteration. Whereas, in the case of MNDS, once certain nodes coalesce to form a cluster, most of the work is done by the root of the cluster.

Processors p	PD		MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	44.37	1.0	5.53	1.0	6.46	1.0
2	30.29	1.5	3.60	1.5	4.01	1.6
3	24.71	1.8	2.86	1.9	3.14	2.1
4	21.85	2.0	2.56	2.2	2.77	2.3
5	20.36	2.2	2.36	2.3	2.52	2.6
6	19.72	2.3	2.07	2.7	2.25	2.9
7	18.76	2.4	1.96	2.8	2.12	3.0
8	18.65	2.4	1.91	2.9	2.07	3.1
9	18.18	2.4	2.15	2.6	2.09	3.1
10	18.34	2.4	2.32	2.4	2.00	3.2
No of iterations	14		4		3	

Table 7.1: Graph with 50 nodes and 500 edges

Processors p	PD		MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	285.32	1.0	23.63	1.0	11.78	1.0
2	182.48	1.6	14.52	1.6	6.76	1.7
3	143.93	2.0	10.79	2.2	5.05	2.3
4	123.51	2.3	9.04	2.6	4.47	2.6
5	111.95	2.5	7.93	3.0	4.24	2.8
6	106.08	2.7	6.18	3.8	3.34	3.5
7	101.28	2.8	6.19	3.8	3.22	3.7
8	98.89	2.9	6.50	3.6	3.03	3.9
9	98.59	2.9	5.99	3.9	3.02	3.9
10	98.32	2.9	5.51	4.3	2.88	4.1
No of iterations	31		6		2	

Table 7.2: Graph with 80 nodes and 1000 edges

Processors p	PD		MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	432.47	1.0	24.13	1.0	21.93	1.0
2	267.72	1.6	14.17	1.7	12.66	1.7
3	207.54	2.1	10.77	2.2	9.20	2.4
4	175.20	2.5	9.01	2.7	7.47	2.9
5	158.42	2.7	8.03	3.0	6.47	3.4
6	148.28	2.9	7.41	3.3	6.33	3.5
7	141.05	3.1	6.94	3.5	5.47	4.0
8	135.28	3.2	6.99	3.5	5.20	4.2
9	132.68	3.3	6.81	3.5	5.07	4.3
10	130.87	3.3	8.67	2.8	5.04	4.3
No of iterations	30		3		3	

Table 7.3: Graph with 100 nodes and 800 edges

Processors p	PD		MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	1356.65	1.0	76.98	1.0	57.29	1.0
2	811.60	1.7	43.36	1.8	32.19	1.8
3	623.14	2.2	31.69	2.4	24.26	2.4
4	521.60	2.6	22.81	3.4	18.99	3.0
5	465.93	2.9	20.70	3.7	16.69	3.4
6	424.52	3.2	18.72	4.1	15.08	3.8
7	400.84	3.4	17.33	4.4	13.91	4.1
8	379.04	3.6	18.30	4.2	10.79	5.3
9	378.98	3.6	16.75	4.6	9.57	6.0
10	367.90	3.7	15.25	5.0	8.48	6.8
No of iterations	40		4		3	

Table 7.4: Graph with 130 nodes and 2500 edges

Processors p	PD		MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	1611.95	1.0	128.17	1.0	82.11	1.0
2	934.30	1.7	71.15	1.8	44.79	1.8
3	695.11	2.3	51.86	2.5	32.36	2.5
4	573.44	2.8	41.17	3.1	26.73	3.1
5	499.15	3.2	43.41	3.0	22.31	3.7
6	452.22	3.6	37.62	3.4	21.12	3.9
7	418.11	3.9	37.37	3.4	19.13	4.3
8	398.47	4.0	26.76	4.8	20.61	4.0
9	377.22	4.3	30.35	4.2	17.02	4.8
10	367.70	4.4	25.36	5.1	17.45	4.7
No of iterations	41		5		3	

Table 7.5: Graph with 160 nodes and 3500 edges

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
3	545.80	2.8	388.02	2.8
4	425.92	3.5	302.05	3.5
5	356.91	4.2	251.77	4.2
6	328.84	4.6	217.63	4.9
7	268.51	5.6	196.64	5.4
8	229.82	6.5	186.40	5.7
9	221.38	6.8	166.57	6.4
10	187.34	8.0	168.44	6.3
11	246.17	6.1	197.10	5.4
12	211.36	7.1	156.83	6.8
13	215.67	7.0	119.04	9.0
14	193.97	7.7	116.13	9.2
15	272.29	5.5	166.21	6.4
No of iterations	7		4	

Table 7.6: Graph with 500 nodes and 12000 edges

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	9257.07	1.0	3830.57	1.0
2	4756.11	1.9	1978.63	1.9
3	3059.46	3.0	1349.56	2.8
4	2357.17	3.9	1084.92	3.5
5	1934.54	4.8	841.72	4.6
6	1644.51	5.6	729.10	5.3
7	880.77	10.5	673.42	5.7
8	866.33	10.7	612.95	6.2
9	1006.72	9.2	534.53	7.2
10	740.45	12.5	521.83	7.3
11	707.61	13.1	463.58	8.3
12	573.84	16.1	486.00	7.9
13	679.79	13.6	375.95	10.2
14	626.13	14.8	373.76	10.2
15	748.00	12.4	334.32	11.5
No of iterations	5		3	

Table 7.7: Graph with 700 nodes and 25000 edges

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	7788.36	1.0	6092.57	1.0
2	4059.64	1.9	3109.96	2.0
3	2962.46	2.6	2112.22	2.9
4	2282.27	3.4	1615.84	3.8
5	1882.29	4.1	1324.02	4.6
6	1598.59	4.9	1123.69	5.4
7	1410.49	5.5	910.25	6.7
8	1179.37	6.6	951.54	6.4
9	773.70	10.1	672.13	9.1
10	1013.09	7.7	632.82	9.6
11	777.86	10.0	575.55	10.6
12	888.68	8.8	622.80	9.8
13	803.98	9.7	493.71	12.3
14	655.77	11.9	530.31	11.5
15	773.81	10.1	505.55	12.1
No of iterations	5		4	

Table 7.8: Graph with 1000 nodes and 25000 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
3	6231.98	4.1	4057.16	2.8
4	5896.98	4.3	3005.84	3.8
5	5011.89	5.1	2038.12	5.6
6	3751.74	6.8	1984.88	5.7
7	3436.68	7.4	1829.64	6.2
8	3063.71	8.3	1676.71	6.8
9	2463.64	10.4	1508.53	7.5
10	2191.07	11.7	1231.60	9.2
11	2102.92	12.1	1311.31	8.7
12	1929.91	13.2	1182.04	9.6
13	2804.87	9.1	1107.76	10.3
14	2208.71	11.6	1032.73	11.0
15	1812.91	14.1	962.72	11.8
No of iterations	5		3	

Table 7.9: 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
3	12575.27	2.9	11304.75	3.6
4	9385.40	3.9	10877.06	3.7
5	7737.92	4.8	7298.14	5.6
6	6267.44	5.9	6153.18	6.6
7	5113.30	7.2	5355.38	7.6
8	5209.42	7.1	4775.39	8.5
9	4404.56	8.4	4358.10	9.3
10	4181.29	8.9	3973.10	10.2
11	4146.25	8.9	3335.38	11.1
12	3819.22	9.7	3137.52	11.8
13	3264.70	11.3	2961.82	12.5
14	3210.79	11.5	3009.98	12.3
15	3895.15	9.5	2826.16	13.1
No of iterations	3		3	

Table 7.10: Graph with 2000 nodes and 30000 edges

CHAPTER 8

INTEGRATED VLSI LAYOUT COMPACTION AND WIRE BALANCING

Layout compaction is an important final step in VLSI physical design process. Compaction is very critical for full-custom layouts, especially for high performance designs. The goal of compaction is to minimize the total layout area without violating any design rules. The process of compaction is very well understood. Complete details of different types of compaction and their relative advantages and disadvantages may be found in [96], [147], [173]. They also contain an extensive list of references relating to compaction.

As mentioned before, Integrated Layout Compaction and Wire Balancing problem is one of the many problems that can be formulated as a dual transshipment problem and solved using one of the algorithms presented in the previous chapters. In this chapter, we will elaborate on the layout compaction and wire balancing problem and show how our algorithms can be used to efficiently solve this problem.

In *layout compaction* one starts with an initial layout and seeks to achieve a final mask layout (without changing the topology) which has minimum chip area and is consistent with the design rules. Invariance of network topology is required in order not to render the previous steps of placement and routing obsolete. At the end of layout compaction relative positions of all the circuit elements will be available. Changing the positions of these elements (to be precise, those elements which lie on a longest path between the chip boundaries) will result in increased chip width. But the positions of the others could be varied without causing design rule violations and yet maintaining compacted chip width. Whereas in *wire balancing*, one seeks to achieve minimum overall wire length by adjusting the positions of the elements without violating any design rules, *integrated layout*

compaction and wire balancing aims at minimizing wire length without increasing the area of the compacted layout. In other words, integrated layout compaction and wire balancing achieves minimum total wire length by adjusting the positions of only those elements which do not lie on the longest paths mentioned above.

8.1 The Constraint Graph Approach

The constraint graph approach to the layout compaction and wire balancing problem proceeds as follows. From the initial layout a graph G = (V, E), called the *constraint graph*, is constructed. We assume that the layout is Manhattan, i.e., edges of circuit elements are either horizontal or vertical. Each node of G represents a circuit element or a group of circuit elements that are physically connected. Each node v_i is associated with a variable y_i representing the position of the corresponding circuit element. In the following the circuit element corresponding to node v_i will also be referred to as v_i. In G, there is an edge between two nodes, if there is a design rule constraint between the corresponding elements. There are three types of constraints: *minimum, maximum* and *equality* constraints.

A minimum constraint of the type $y_j - y_i \geq a$ states that v_i is to the left of v_j and there is a minimum spacing requirement of 'a' units between them. This constraint can also be stated as $y_i - y_j \leq -a$ and is represented in G by an edge (v_j, v_i) directed from v_j to v_i with an associated token m_{ji} of value $-a$.

A maximum constraint of the form $y_j - y_i \leq b$ or equivalently $y_i - y_j \geq -b$ is represented by an edge (v_i, v_j) directed from v_i to v_j with token m_{ij} of value "b".

An equality constraint can be regarded as a pair of minimum and maximum constraints. Thus an equality constraint will be represented by a pair of oppositely directed edges both with zero token.

Two special nodes, called the source (v_s) and the sink (v_t), are used to represent the

right most boundary and the left most boundary of the layout, respectively. Circuit elements which correspond to nodes with no outgoing edges could be placed at the left boundary, and so we add to G edges directed from each one of these nodes to the sink v_t. These edges have zero token. For a similar reason, we add to G zero-token edges directed from the source v_s to the nodes with no incoming edges. The additional edges so added ensure that the circuit elements will not be moved beyond the left and the right boundaries. These edges play a key role in the wire balancing phase.

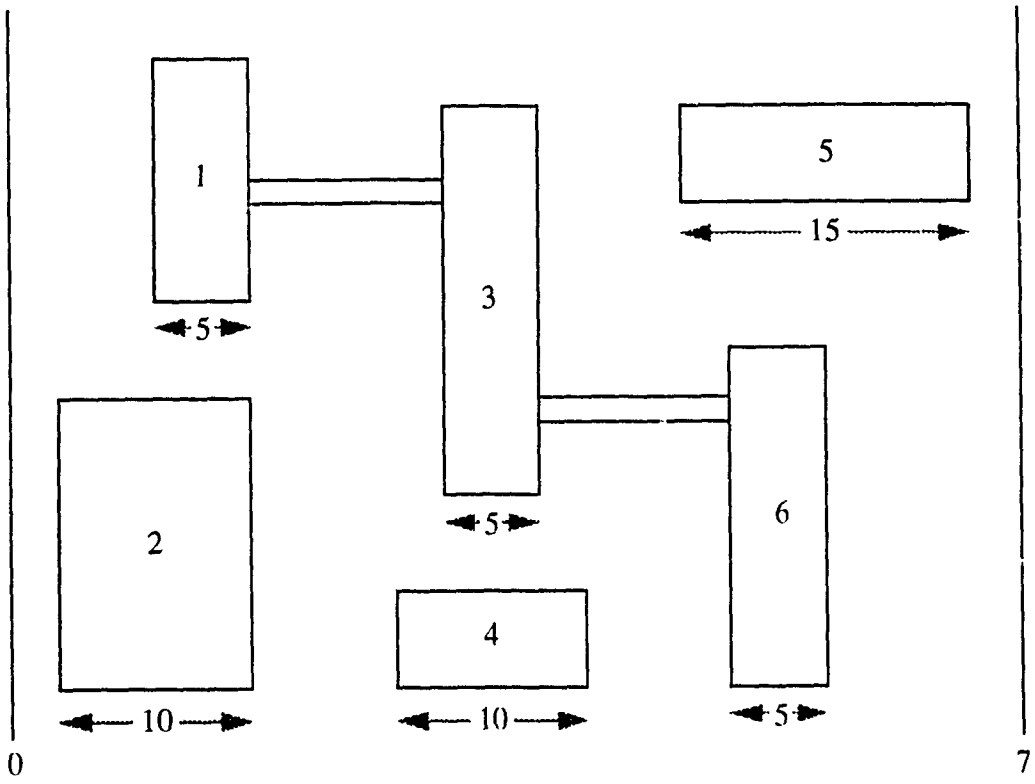


Figure 8.1 An example of a layout

Besides tokens, edges are also associated with weights to indicate the relative costs of wires. We derive node weights from edge weights as follows. Let w_{ij} denote the weight of the edge (v_i, v_j) directed from v_i to v_j. Then the weight w_i of node v_i is given

$$w_i = \sum_j w_{ij} - \sum_j w_{ji}$$

Here we assume that $w_{ij} = 0$ if there is no wire between v_i and v_j. Thus a negative node weight indicates that moving the node to the right will decrease the overall wire length and a positive node weight indicates that moving a node to the right will increase the overall wire length.

$y_0 - y_1 \leq 0$

$y_0 - y_2 \leq 0$

$y_1 - y_3 \leq -10$

$y_2 - y_3 \leq -15$

$y_2 - y_4 \leq -15$

$y_3 - y_5 \leq -10$

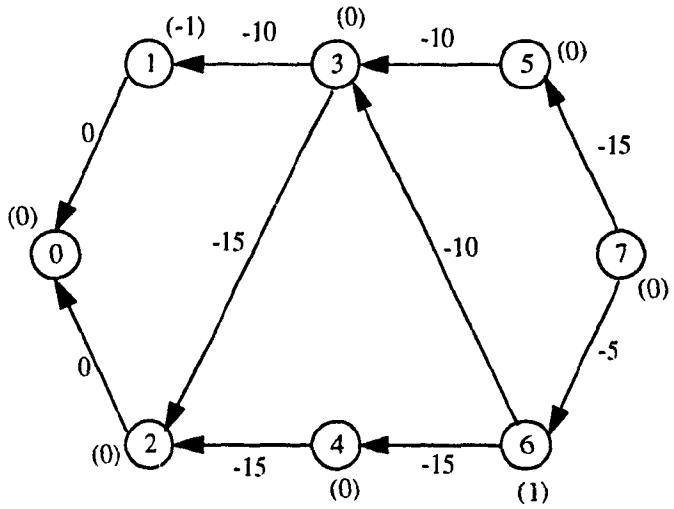
$y_3 - y_6 \leq -10$

$y_4 - y_6 \leq -15$

$y_5 - y_7 \leq -15$

$y_6 - y_7 \leq -5$

(a) Minimum spacing constraints



(b) Constraint graph

Figure 8.2 Spacing Constraints and the Constraint Graph for the layout in Fig. 8.1.

As an example, consider the layout shown in Figure 8.1. The cell numbers are shown inside the cells and cell widths are as shown. We assume that the layout is to be compacted in the x-direction. We also assume, for the sake of simplicity that the minimum spacing requirements between any two cells are 5 units. There is no minimum spacing requirements between a cell and chip boundary. Further, we assume that each cell's position is represented by its left x-coordinate. Therefore, to calculate minimum spacing between two cells, the cell width of left cell has to be added to 5. We also assign a node

weight of -1 to node 1 and 1 to node 6. All other nodes will have a weight of zero. Now, the minimum spacing constraints and its corresponding constraint graph are as shown in Figure 8.2.

8.2 Layout Compaction

Let Y denote the vector of y_i's, M^t the vector of m_{ij}'s and W, the vector of node weights. Let A be the incidence matrix of G. Then in layout compaction we seek to obtain a $Y \geq 0$ such that

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

and that the difference between the largest and the smallest y_i's is as small as possible. Thus we can formulate this problem as an LP problem as follows.

Minimize: $y_s - y_t$

subject to

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

$$Y \geq 0.$$

Assuming that the position of the sink is at zero coordinate, the above formulation can be written as

Minimize: y_s

subject to

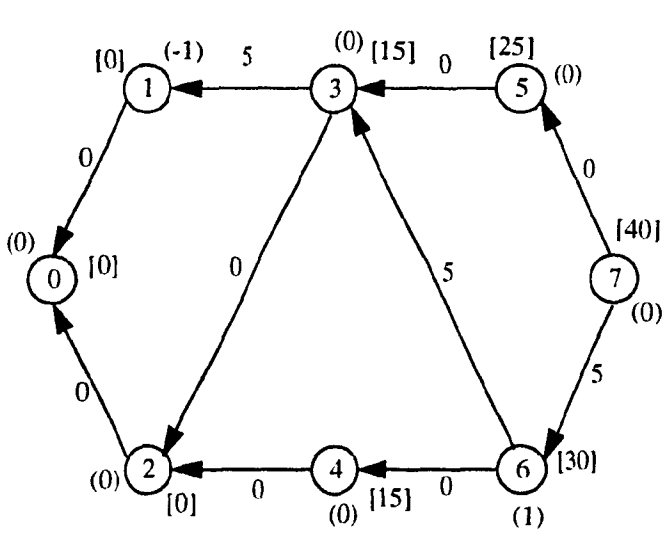
$$A^t Y \geq -M^t$$

$$Y \geq 0. \qquad (8.1)$$

The above formulation of the layout compaction is clearly a special case of the dual transshipment problem. Our algorithm FEASIBLE in fact solves the above layout compaction problem if there are no negative-token directed circuits in G. If we assume that there are no such circuits, then the y_i-values obtained at the end of FEASIBLE repre-

sent the positions of the different circuit elements. Each node v_i with $y_i = 0$ will be at the left boundary and each v_i with the maximum y_i will be at the right boundary. It can be shown that the maximum y_i-value in fact is the length of the most negative token directed path in G from the source to the sink. In traditional approaches, such a path in fact corresponds to a longest path from the sink v_t to the source v_s.

Figure 8.3 shows the graph after applying Algorithm FEASIBLE on the constraint graph of Figure 8.2 and the corresponding cell positions. Also, cells 1 and 6 can be moved to the right upto 5 units without increasing the width of the total layout. This can be calculated by looking at the residual tokens of the incoming edges at each node.



(a) Graph After Compaction

$y_0 = 0$

$y_1 = 0$

$y_2 = 0$

$y_3 = 15$

$y_4 = 15$

$y_5 = 25$

$y_6 = 30$

$y_7 = 40$

(b) Cell Positions

Figure 8.3 The Layout Compaction

8.3 Wire Balancing

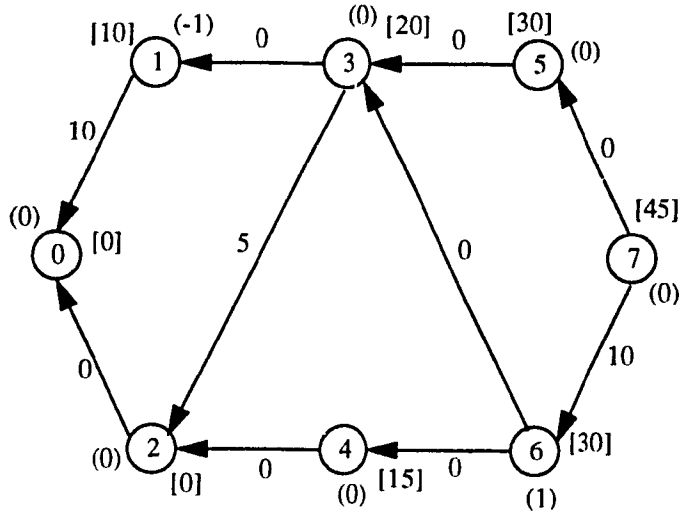
The wire balancing problem where we seek to find a Y which minimizes $\sum_i w_i y_i$ can be formulated as the following linear programming problem.

Minimize: $W^t Y$

subject to

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

$$Y \geq 0. \tag{8.2}$$



(a) Graph After Wire Balancing

$y_0 = 0$

$y_1 = 10$

$y_2 = 0$

$y_3 = 20$

$y_4 = 15$

$y_5 = 30$

$y_6 = 30$

$y_7 = 45$

(b) Cell Positions

Figure 8.4 The Wire Balancing

Note that layout compaction (8.1) is a special case of the wire balancing problem. Algorithm FEASIBLE determines y_i's which satisfy $A^t Y \geq -M^t$ and so this algorithm determines a feasible solution for both (8.1) and (8.2). Thus algorithm FEASIBLE serves as a link between the layout compaction and wire balancing problems.

If the graph in Figure 8.2 is solved using the above formulation, then the resulting graph and the corresponding cell positions will be as shown in Figure 8.4. Notice the width of the chip is not minimum even though the total wire length is minimum here.

8.4 Integrated VLSI Layout Compaction And Wire Balancing

Let $y_i = \lambda_i$ at the end of Algorithm FEASIBLE. Recall that this algorithm modifies the edge tokens at each step. When all the edge tokens are non-negative (that is $\lambda_i - \lambda_j + m_{ij} \geq 0$), the algorithm terminates. Interestingly, at termination the tokens of the edges on the most negative token directed path from v_s to v_t will all be zero. In other words, for every (i, j) on such paths $y_i - y_j + m_{ij} = 0$. So by a simple traversal of G starting from v_s, we can identify all the nodes on these most negative token paths. Let S_l denote the set of these nodes.

As we mentioned before, our aim in integrated compaction and wire balancing is to keep each node $v_i \in S_l$ at $y_i = \lambda_i$ and adjust the y-values of those nodes not in S_l so that total wire length can be minimized without increasing the area of the layout. Thus we have the following formulation of the integrated compaction and wire balancing problem.

Minimize: $W^t Y$

subject to

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

$$Y \geq 0. \tag{8.3}$$

$$y_i = \lambda_i, \text{ for } v_i \in S_l, \tag{8.4}$$

$$y_i \geq 0, \text{ for all other } v_i .$$

But, as seen above constraint (8.4) can be replaced by:

$$y_i - y_j + m_{ij} = 0 \tag{8.5}$$

for every edge (v_i, v_j) on a most negative token path from v_s to v_t.

Now it is a simple matter to represent each of the above equality constraints by adding to G oppositely oriented edges with tokens m_{ij} and m_{ij} between v_i and v_j , and between v_j and v_i and then proceed with the solution of the resulting optimization problem.

However, such an approach will result in a significant increase in the size of the graph. In fact, we can considerably reduce the size of the graph as discussed below.

Constraint (8.5) suggests that in any new solution of the wire balancing problem,

$$y_i = \lambda_i + k, \text{ for } v_i \in S_l$$

where k is a fixed constant. We can take advantage of this property as follows.

We construct a new graph G′ by replacing the nodes in S_l by a single new node, say, v_l, and then removing all the edges connecting the nodes in S_l. Consider now an edge (v_i, v_j) in G. If $v_i \notin S_l$ and $v_j \notin S_l$, G′ will have an edge (v_i, v_j) with token m_{ij}. If $v_i \in S_l$, then G′ will have an edge (v_l, v_j) with token $(\lambda_i + m_{ij})$. If $v_j \in S_l$, then G′ will have an edge (v_i, v_l) with token $(-\lambda_j + m_{ij})$. If after this contruction, there are parallel edges, in G′, between two nodes then we can remove all of them except the one with the smallest token. Let the new graph be denoted by G″. In G″ the weight of every node $v_i \notin S_l$ will be equal to w_i. For the new node v_l representing S_l, the weight w_l will be given by

$$w_l = \sum_{v_i \in S_l} w_i.$$

We now have the following formulation of the integrated compaction and wire balancing problem, where w_i, y_i, m_{ij} refer to the quantities defined for G″.

$$Minimize \sum_i w_i y_i$$

subject to

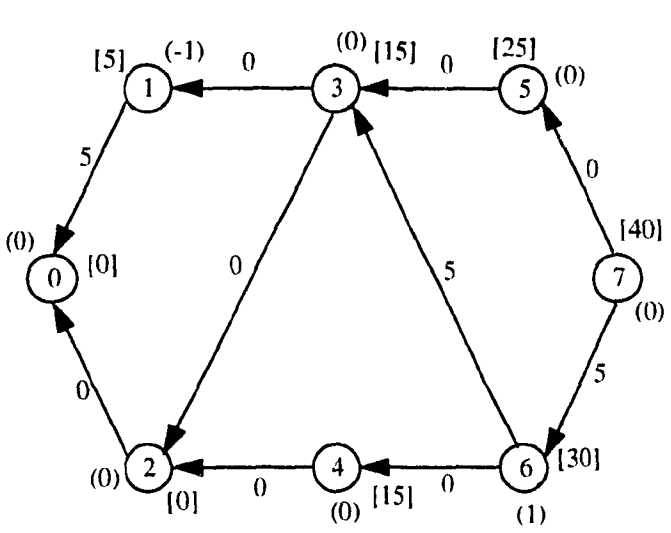
$$A^t Y \geq -M^t$$

$$Y \geq 0, \tag{8.6}$$

where A is the incidence matrix of G″.

Figure 8.5 shows the graph after solving it as a integrated layout compaction and wire balancing problem. Notice here the chip width is minimum, but total wire length is

not the minimum possible.



(a) Graph After Integrated Compaction and Wire Balancing

$y_0 = 0$

$y_1 = 5$

$y_2 = 0$

$y_3 = 15$

$y_4 = 15$

$y_5 = 25$

$y_6 = 30$

$y_7 = 40$

(b) Cell Positions

Figure 8.5 The Integrated Layout Compaction and Wire Balancing

8.5 Experimental Results

The wire balancing problem can be formulated as a dual transshipment problem. So, from our results in the previous chapter, we can conclude that CBDS has better performance than MNDS when used to solve the wire balancing problem.

To test the performance of these two algorithms for the integrated layout compaction and wire balancing problem, we have applied them on constraint graphs derived from large industrial designs supplied by Cadence Design Systems. In integrated layout compaction and wire balancing problem, the positions of the nodes which lie on longest paths between the source and the sink are fixed after the layout compaction phase. This can be done in two ways. The first way of fixing the positions of the nodes on longest paths is by introducing a 0-token edge directed from the sink to the source. Tables 8.1 - 8.5 give

results obtained in this way for different graphs. The second way of fixing the positions of the nodes on longest paths is by coalescing them into one node. Tables 8.6 - 8.10 give results obtained in this way. Also, for large graphs, longest path condensation achieves about 30% reduction in computation time. These results show that MNDS has a better performance than CBDS. This could be explained as follows.

The integrated layout compaction and wire balancing problem consists two steps. First, layout compaction is achieved. In the second phase, all the nodes which lie on longest paths between the source and the sink are coalesced into a single cluster C. The number of such nodes will usually be large. In the subsequent steps of the second phase of MNDS, C will become part of a big cluster S which contains all the negative nodes in the graph. The only nodes which are not part of S are the non-negative nodes which are not part of C. On the other hand, in the CBDS algorithm, overlapping clusters are not combined, they are treated as distinct clusters. As a result, cluster firings during the cluster optimization step of CBDS will cause repeated cluster expansions and cluster reductions, resulting in considerable overhead cost.

We have also summarized in Table 8.11 the savings in layout area and minimum total wire length achieved by wire balancing as well as integrated layout compaction and wire balancing.

8.6 Summary

In this chapter, we have shown that the wire balancing problem can be formulated as a dual transshipment problem (DTP) and that the layout compaction problem is a special instance of the DTP. In fact, layout compaction can be accomplished by applying Algorithm FEASIBLE to test feasibility of the DTP. We have also introduced and presented a unified formulation of the integrated layout compaction and wire balancing problem. We have shown that the integrated layout compaction and wire balancing problem

can be solved in two steps:

1 First apply Algorithm FEASIBLE to achieve compaction.

2 Then solve DTP on a condensed graph obtained by coalescing the nodes on longest paths (between the source and the sink) into a single node.

We have also tested the CBDS and MNDS algorithms of Chapters 5 and 6 to evaluate their performance for the integrated layout compaction and wire balancing problem. Our results show that MNDS has better performance in this case than CBDS.

We have also summarized in a table the savings in layout area and wire length achieved by integrated layout compaction and wire balancing. For large graphs, these savings are significant.

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	38.22	1.0	66.36	1.0
2	23.49	1.6	37.15	1.8
3	17.29	2.2	25.69	2.6
4	14.43	2.6	21.37	3.1
5	13.32	2.9	18.80	3.5
6	11.48	3.3	17.16	3.9
7	9.69	3.9	15.68	4.2
8	9.57	4.0	12.79	5.2
9	8.08	4.7	13.84	4.8
10	7.94	4.8	11.26	5.9

Table 8.1: Graph with 149 nodes

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	24.84	1.0	49.93	1.0
2	14.88	1.7	28.68	1.7
3	11.46	2.2	21.13	2.4
4	9.25	2.7	17.36	2.9
5	8.99	2.8	15.06	3.3
6	8.31	3.0	13.16	3.8
7	7.55	3.3	12.28	4.1
8	6.93	3.6	11.65	4.3
9	5.17	4.8	10.56	4.7
10	5.10	4.9	9.94	5.0

Table 8.2: Graph with 157 nodes

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	9594.12	1.0	14448.54	1.0
2	4149.77	2.3	7502.77	1.9
3	2430.60	3.9	5103.73	2.8
4	1989.72	4.8	3987.87	3.6
5	1719.53	5.6	3226.58	4.5
6	1325.65	7.2	2752.19	5.2
7	1313.02	7.3	2354.14	6.1
8	1149.14	8.3	2146.71	6.7
9	924.03	10.4	1915.57	7.5
10	911.42	10.5	1779.97	8.1
11	847.72	11.3	1662.91	8.7
12	840.76	11.4	1559.26	9.3
13	808.08	11.9	1485.32	9.7
14	754.63	12.7	1409.30	10.3
15	763.76	12.6	1338.17	10.8

Table 8.3: Graph with 1265 nodes

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	13900.81	1.0	76979.97	1.0
2	7647.31	1.8	41145.48	1.9
3	4960.28	2.8	27205.57	2.8
4	4040.12	3.4	21460.55	3.6
5	3212.87	4.3	17770.85	4.3
6	2698.88	5.2	14677.87	5.2
7	2620.52	5.3	13193.71	5.8
8	2438.55	5.7	11830.17	6.5
9	2220.26	6.3	10676.41	7.2
10	1969.57	7.1	9848.20	7.8
11	1786.19	7.8	9192.89	8.4
12	1905.69	7.3	8600.66	9.0
13	1408.92	9.9	8178.21	9.4
14	1767.32	7.9	7849.30	9.8
15	1630.05	8.5	7375.50	10.4

Table 8.4: Graph with 1612 nodes

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	39567.42	1.0	49545.82	1.0
2	18139.42	2.2	26076.75	1.9
3	13079.62	3.0	17660.34	2.8
4	9078.71	4.4	14033.11	3.5
5	8513.56	4.6	11281.16	4.4
6	7039.69	5.6	9382.05	5.2
7	5867.42	6.7	9222.33	5.4
8	5360.34	7.4	7451.10	6.6
9	5088.48	7.8	6593.45	7.5
10	4594.10	8.6	6083.19	8.1
11	4220.74	9.4	5689.40	8.7
12	4110.56	9.6	5785.26	8.6
13	3996.97	9.9	4916.21	10.1
14	3816.34	10.4	4504.17	11.0
15	3647.82	10.8	4630.45	10.7

Table 8.5: Graph with 2087 nodes

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	16.77	1.0	27.29	1.0
2	10.01	1.7	13.59	2.0
3	8.04	2.1	11.37	2.4
4	6.58	2.5	9.34	2.9
5	5.99	2.8	8.34	3.3
6	5.84	2.9	6.98	3.9
7	5.35	3.1	5.38	5.1
8	5.06	3.3	5.97	4.6
9	4.93	3.4	5.85	4.7
10	4.92	3.4	5.04	5.4

Table 8.6: Graph with 149 nodes after longest path condensation

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	9.94	1.0	21.23	1.0
2	5.80	1.7	11.81	1.8
3	4.32	2.3	8.39	2.5
4	3.69	2.7	6.80	3.1
5	3.39	2.9	5.84	3.6
6	3.18	3.1	5.28	4.0
7	2.65	3.8	4.80	4.4
8	2.81	3.5	4.49	4.7
9	2.22	4.5	4.31	4.9
10	2.10	4.7	4.14	5.1

Table 8.7: Graph with 157 nodes after longest path condensation

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	6246.28	1.0	11299.64	1.0
2	2940.98	2.1	5737.28	2.0
3	2099.63	3.0	3854.87	2.9
4	1654.00	3.8	2962.38	3.8
5	1206.52	5.2	2401.74	4.7
6	1169.54	5.3	2050.52	5.5
7	1131.48	5.5	1783.32	6.3
8	949.17	6.6	1597.68	7.1
9	793.67	7.9	1439.47	7.8
10	901.98	6.9	1331.03	8.5
11	663.51	9.4	1242.09	9.1
12	609.81	10.2	1163.48	9.7
13	653.85	9.6	1107.19	10.2
14	741.68	8.4	1054.64	10.7
15	724.64	8.6	1016.64	11.1

Table 8.8: Graph with 1265 nodes after longest path condensation

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	10013.66	1.0	57714.52	1.0
2	4882.85	2.1	29156.66	2.0
3	3055.94	3.3	19757.81	2.9
4	2512.66	4.0	15014.36	3.8
5	2353.02	4.3	12013.04	4.8
6	1922.75	5.2	10110.98	5.7
7	1731.23	5.8	8913.39	6.5
8	1567.94	6.4	8090.20	7.1
9	1415.48	7.1	7347.97	7.9
10	1487.65	6.7	6817.72	8.5
11	1320.82	7.6	6372.41	9.1
12	1389.09	7.2	5999.99	9.6
13	1175.26	8.5	5730.26	10.1
14	1176.46	8.5	5466.35	10.6
15	1307.63	7.7	5263.27	11.0

Table 8.9: Graph with 1612 nodes after longest path condensation

Processors p	MNDS		CBDS	
	Time in Seconds	Parallel Speed up	Time in Seconds	Parallel Speed up
1	33738.35	1.0	36366.62	1.0
2	16606.27	2.0	18972.09	1.9
3	11830.75	2.9	12811.18	2.8
4	9313.02	3.6	9621.03	3.8
5	8157.26	4.1	7903.62	4.6
6	6807.90	5.0	7099.16	5.1
7	5655.07	6.0	5900.93	6.2
8	5126.10	6.6	5155.67	7.1
9	5162.44	6.5	4917.64	7.4
10	4557.98	7.4	4169.96	8.7
11	4453.89	7.6	4370.84	8.3
12	4217.13	8.0	3891.69	9.3
13	4089.29	8.3	3716.85	9.8
14	4075.00	8.3	3691.56	9.9
15	3881.05	8.7	3853.43	9.4

Table 8.10: Graph with 2087 nodes after longest path condensation

	No of Nodes N	Objective x 10^8	Improvement %	Chip Width	Improvement %
Layout Compaction	149	35.65788	–	49600	–
Wire Balancing		33.35350	6.46	49600	0
Integrated Layout Compaction & Wire Balancing		33.35350	6.46	49600	0
Layout Compaction	157	8.270312	–	123800	–
Wire Balancing		6.396859	22.65	128300	–3.63
Integrated Layout Compaction & Wire Balancing		6.654659	19.54	123800	0
Layout Compaction	1265	11348.29	–	331450	–
Wire Balancing		5636.508	50.33	370250	–11.71
Integrated Layout Compaction & Wire Balancing		6239.136	45.02	331450	0
Layout Compaction	1612	52485.97	–	321000	–
Wire Balancing		28140.84	46.38	321000	0
Integrated Layout Compaction & Wire Balancing		28140.84	46.38	321000	0
Layout Compaction	2087	378.7866	–	1016000	–
Wire Balancing		307.5949	18.79	1025500	–0.94
Integrated Layout Compaction & Wire Balancing		307.7869	18.74	1016000	0

Table 8.11: Savings in Chip Width and Total Wire Length

CHAPTER 9

SUMMARY AND PROBLEMS FOR FUTURE STUDY

In this concluding chapter, we present a summary of the work reported in this thesis and suggest a few problems for further study.

9.1 Summary

Network optimization refers to the general class of optimization problems defined on graphs and networks. The transshipment and the dual transshipment problems generalize several of the network optimization problems which include the shortest paths, max-flow and matching algorithms. These problems can be used to formulate and solve a large class of industrial and engineering problems. As the complexity and sizes of these problems increase, there is a growing demand for more computing power. Parallel computing is one of the ways to meet this demand. This has motivated our work in this thesis on parallel network optimization.

Chapters 2 - 7 are concerned with the design, implementation and experimental evaluation of three parallel algorithms - one for the transshipment problem and two for its dual. Chapter 8 is concerned with the application of our algorithms in VLSI layout compaction and wire balancing problems. Our algorithms extensively use the notion of node/cluster firings and results from the theory of marked graphs.

In Chapter 2, we reviewed the transshipment problem and its dual as well as certain basic results and definitions which are used in the development of our algorithms.

In Chapter 3, three basic network optimization algorithms and their parallel implementations were discussed. They are: Algorithm FEASIBLE to test feasibility of the dual transshipment problem, Bellman-Ford-Moore shortest path algorithm and Goldberg-Tarjan max-flow algorithm. These algorithms serve as building blocks in our parallel algo-

rithms for solving the transshipment and dual transshipment problems.

In Chapter 4, we discussed the details of the primal-dual approach to the transshipment problem as well as the issues which are encountered in its parallel implementation. The primal-dual approach involves repeated applications of the max-flow and shortest path algorithms discussed in Chapter 3. We have shown that Algorithm FEASIBLE can be adapted for initialization of the primal-dual method. This approach to initialization does not require constructing an auxiliary graph unlike the traditional approach to primal-dual initialization. This is quite attractive from the point of view of a parallel implementation.

The network dual simplex method is known to be a very efficient approach for solving the dual transshipment problem, but it does not offer much scope for parallelisation. In Chapter 4, we developed a new approach to the dual transshipment problem. This method, called the Modified Network Dual Simplex (MNDS) method, has several features which make it amenable for an efficient parallel implementation. Unlike the traditional network dual simplex method, MNDS does not move from one basic solution to another. MNDS employs Algorithm FEASIBLE to test feasibility and then involves repeated applications of the shortest path algorithm and concurrent pivot operations. Performing concurrent pivots efficiently without destroying feasibility, and transforming a feasible solution to a basic feasible solution through shortest path computations without decreasing the objective value are the distinguishing features of MNDS.

In Chapter 6, we first established a new characterisation of the optimum solutions to the dual transshipment problem in terms of certain clusters rooted at negative-weighted nodes. Based on this characterisation, we then developed a novel parallel algorithm for the dual transshipment problem. This new algorithm, called the Cluster-Based Dual Simplex (CBDS) method, employs Algorithm FEASIBLE to test feasibility and involves repeated applications of a cluster optimization algorithm (node/cluster firings) and concurrent pivots. CBDS significantly differs from MNDS from the way a feasible solution is trans-

formed to a basic feasible solution. The cluster optimization algorithm used in CBDS resembles Prim's algorithm for constructing a min-cost spanning tree.

We have reported in Chapter 7, results of an experimental evaluation of the three parallel algorithms developed in Chapters 4 - 6. The evaluation has been carried out in a shared memory parallel programming environment.

Finally, in Chapter 8, we considered application of our parallel network optimization algorithms in VLSI layout compaction and wire balancing problems. We first showed that the wire balancing problem can be formulated as a dual transshipment problem and that layout compaction is a special instance of the dual transshipment problem. We then introduced the integrated layout compaction and wire balancing problem and presented a unified formulation of this problem in terms of the dual transshipment problem. We have also reported results of a comparative evaluation of the CBDS and MNDS algorithms with respect to their suitability for the wire balancing and the integrated layout compaction and wire balancing problems.

9.2 Problems for Future Study

Our work in this thesis suggest a few problems for further study:

- The network primal dual method is known to be a very efficient sequential algorithm for solving the transshipment problem. This does not appear to be the case from the point of view of a parallel implementation. As we have pointed out in Chapter 7, our choice of Goldberg-Tarjan algorithm for the max-flow problem may be the source of the poor performance of the network primal dual method. It will be interesting to study its performance using Dinic's and MPM's algorithms for max-flow computations and compare this performance with that of the parallel network simplex method discussed in [131], [154].

- The CBDS and MNDS algorithms have been designed to incorporate features which

make them amenable for efficient parallel implementations. However, we believe that even as sequential algorithms they are comparable to the network dual simplex method. Implementation of these algorithms on a uniprocessor might provide insight on their relative merits.

- We believe that the network dual simplex method is not suited for parallel implementation. However, a parallel implementation of this method is needed to establish its merit relative to our parallel algorithms based on MNDS and CBDS.

- Implementing the CBDS and MNDS algorithms with upper bounds on the y-values is another direction of work worth pursuing.

- Our implementations have assumed a shared-memory programming environment. Adaptability of CBDS and MNDS to suit implementations under the message-passing and workstation-based models [156] requires investigation.

- The structure of the optimum solutions as presented in Theorem 6.1 has not been fully used by CBDS as a test for optimality. It will be worth investigating whether this structure would lead to a simpler test for optimality other than the traditional simplex criterion.

REFERENCES

[1] Agarwal, D.P., V.K. Janakiram and R. Mehrotra, "A Randomized Parallel Branch and Bound Algorithm," *Proc. Intl. Conf. on Parallel Processing*, 1988.

[2] Agarwal, P., W.J. Dally, A.K. Ezzat, W.C. Fischer, H.V. Jagdish and A.S. Krishnakumar, "Architecture and Design of the Mars Hardware Accelerator," *Proc. Design Automation Conf.*, 108-113, Jun. 1987.

[3] Agarwal, P., W.J. Dally, W.C. Fischer, H.V. Jagdish, A.S. Krishnakumar and R. Tutundjian, "Mars: A Multiprocessor Based Programmable Accelerator," *IEEE Design and Test*, 28-36, Oct. 1987.

[4] Agarwal, P., "VLSI Computer Aided Design Using Multiprocessing Systems," *Technical Report,* AT&T Bell Labs, Murray Hill, N.J., 1991.

[5] Akers, S.B., C. Joseph and B. Krishnamurthy, "On the Role of Independent Fault Sets in the Generation of Minimal Test Sets," *Proc. IEEE Intl. Test Conf.*, 1100-1107, Oct. 1987.

[6] Akl, S., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Inc., 1989.

[7] Arnold, J.M. and C.T. Terman, "A Multiprocessor Implementation of a Logic-Level Timing Simulator," *Proc. Intl. Conf. on Computer-Aided Design*, 116-118, 1985.

[8] Arvindam, S., V. Kumar and V. Nageshwara Rao, "Floorplan Optimization on Multiprocessors," *Proc. 1989 Intl. Conf. on Computer Design (ICCD-89)*, 1989.

[9] Arvindam, S., V. Kumar and V. Nageshwara Rao, "Efficient Parallel Algorithms for Search Problems: Applications in VLSI CAD," *Proc. Frontiers 90 Conf. on*

Massively Parallel Computation, October 1990.

[10] Arvindam, S., V. Kumar, V. Nageshwara Rao and V. Singh, "Automatic Test Pattern Generatioon on Parallel Processors," *Technical Report*, Computer Science Department, Univ. of Minnesota, May 1990.

[11] Banerjee, P., "The Use of Parallel Processing for VLSI CAD Applications: A Tutorial," *Proceedings International Conference on CAD*, ICCAD-88, 1988.

[12] Banerjee, P., *Parallel Algorithms for VLSI Computer-Aided Design Applications*, Draft, Prentice-Hall, Inc., to be published.

[13] Banerjee, P., M.H. Jones and J.S. Sargent, "Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, 91-106, January 1990.

[14] Barzilai, Z., J.L. Carter, B.K. Rosen and J.D. Rutledge, "HSS - A High Speed Simulator," *IEEE Trans. Computer Aided Design*, 601-617, Jul. 1987.

[15] Belkhale, K.P. and P. Banerjee, "PACE: A Parallel VLSI Circuit Extractor on the Intel Hypercube Multiprocessor," *Proc. Intl. Conf. Computer-Aided Design*, 326-329, Nov. 1988.

[16] Belkhale, K.P. and P. Banerjee, "PACE2: An Improved Parallel VLSI Circuit Extractor with Parametric Extraction," *Proc. Intl. Conf. Computer-Aided Design*, 526-530, Nov. 1989.

[17] Belkhale, K.P. and P. Banerjee, "Parallel Algorithms for VLSI Circuit Extraction." *IEEE Trans. Computer-Aided Design*, 604-618, May 1991.

[18] Bellman, R.E., "On a Routing Problem." *Quart. Appl. Math.*, Vol. 16, 87-90, 1958.

[19] Bertsekas, D.P. and J. Tsitsiklis, "Parallel and Distributed Computations: Numerical Methods," *Prentice Hall*, N.J., 1989.

[20] Bier, G.E. and A.R. Pleszkun, "An Algorithm for Design Rule Checking on a Multiprocessor," *Proc. Design Automation Conference*, 299-303, 1985.

[21] Bixby, R., "Two Applications of Linear Programming," *Proc. Workshop on Parallel Computing of Discrete Optimization Problems*, 1991.

[22] Blank, T., M. Stefik and W. vanCleemput, "A Parallel Bit Map Architecture for DA Algorithms," *Proc. Design Automation Conference*, 837-845, 1981.

[23] Boyer, D.G., "Virtual Grid Compaction Using the Most Recent Layers Algorithm," *Proceedings of the International Conference on Computer-Aided Design*, pp. 92-93, IEEE, 1986.

[24] Boyer, D.G., "Symbolic Layout Compaction Review," *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 383-389, 1988.

[25] Briner, J., "Parallel Mixed Level Simulation of Digital Circuits Using Virtual Time," *Ph.D. Thesis*, Duke Univ., 1990.

[26] Bryant, R., "Data Parallel Switch-Level Simulation," *Proc. Intl. Conf. Computer-Aided Design (ICCAD-88)*, Nov. 1988.

[27] Carlson, E.C. and R.A. Rutenbar, "A Scanline Data Structure Processor for VLSI Geometric Checking," *IEEE Trans. Computer-Aided Design*, 780-794, Sept. 1987.

[28] Carlson, E.C. and R.A. Rutenbar, "Mask Verification on the Connection Machine," *Proc. Design Automation Conference*, 134-140, 1988.

[29] Carlson, E.C. and R.A. Rutenbar, "Design and Performance Evaluation of New

Massively Parallel VLSI Mask Verification Algorithms in Jigsaw," *Proc. Design Automation Conference*, 253-259, 1990.

[30] Casotto, A., F. Romeo and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. CAD-6, No. 5, 838-847, September 1987.

[31] Casotto, A. and A. Sangiovanni-Vincentelli, "Placement of Standard Cells Using Simulated Annealing on the Connection Machine," *Proceedings of the International Conference on Computer-Aided Design,* ICCAD-87, 350-353, 1987.

[32] Chakradhar, S., M.L. Bushnell and V.D. Agarwal, "Towards Massively Parallel Automatic Test Generation," *IEEE Trans. Computer-Aided Design*, 981-994, Sep. 1990.

[33] Chamberlain, R. and M.A. Franklin, "Discrete Event Simulation on Hypercube Architectures," *Proc. Intl. Conf. Computer-Aided Design (ICCAD-88)*, Nov. 1988.

[34] Chandra, S. and J.H. Patel, "Test Generation in a Parallel Processing Environment," *Proc. Intl. Conf. Comp. Design (ICCD-88)*, 11-14, Oct. 1988.

[35] Chandy, K.M. and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Trans. Software Engg.*, SE-5, 44-45, Sep. 1979.

[36] Chandy, K.M. and J. Misra, "Distributed Computation on Graphs," *CACM*, Vol. 25, 833-837, 1982.

[37] Cho, Y.E., "A Subjective Review of Compaction," *Proceedings of the 22nd Design Automation Conference,* ACM/IEEE, 396-404, 1985.

[38] Cho, Y.E., A.J. Korenjak and D.E. Stockton, "FLOSS: An Approach to Automated Layout for High-Volume Designs," *Proceedings of the 14th Design Automation Conference*, ACM/IEEE, 138-141, 1977.

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

[40] Chyan, D.J. and M.A. Breuer, "A Placement Algorithm for Array Processors," *Proceedings of the 20th Design Automation Conference*, ACM/IEEE. 182-188, 1983.

[41] Cohoon, J.P., S.U. Hegde, W.N. Martin and D. Richards, "Distributed Genetic Algorithms for the Floorplan Design Problem," *IEEE Trans. Computer-Aided Design*, 10(4), April 1991.

[42] Comeau, M.A., "Reachability and Sequencing in Marked Graphs and State Graphs: Algorithms Based on Network Programming," Ph.D. thesis, Dept. of Electrical & Computer Engineering, Concordia University, Montreal, June 1986.

[43] 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.

[44] 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, Sept. 1987.

[45] Darema, F., S. Kirkpatrick and V.A. Norton, "Parallel Algorithms for Chip Placement by Simulated Annealing," *IBM J. Res. Dev.*, May 1987.

[46] De, K., B. Ramkumar and P. Banerjee, "Propersyn: A Portable Parallel Algorithm

for Logic Synthesis," *Proc. Intl. Conf. Computer-Aided Design*, Nov. 1992.

[47] DeMello, J.D., J.L. Calvet and J.M. Garcia, "Vectorization and Multitasking of Dynamic Programming in Control: Experiments on a CRAY-2," *Parallel Computing*, 13:261-269, 1990.

[48] Deutsch, J.T. and A.R. Newton, "A Multiprocessor Implementation of Relaxation-Based Electrical Circuit Simulation," *Proc. Design Automation Conf.*, 350-357, 1984.

[49] Dijkstra, E.W., "A Note on Two Problems in Connexion with Graphs," *Numerische Math.*, Vol. 1, 269-271, 1959.

[50] Dinic, E.A., "Algorithm for the Solution of a Problem of Maximum Flow in a Network with Power Estimation," *Soviet Math., Dokl.*, Vol. 11, 1277-1280, 1970.

[51] Duba, P.A., R.K. Roy, J.A. Abraham and W.A. Rogers, "Fault Simulation in a Distributed Environment," *Proc. Design Automation Conference*, Jun. 1988.

[52] Dunn, L.N., "IBM's Engineering Design System Support for VLSI Design and Verification," *IEEE Design and Test*, 30-40, Feb. 1984.

[53] Ferguson, C. and R. Korf, "Distributed Tree Search and its Application to Alpha-Beta Pruning," *Proc. 1988 National Conf. on Artificial Intelligence*, August 1988.

[54] Finkel, R.A. and U. Manber, "DIB - A Distributed Implementation of Backtracking," *ACM Trans. of Programming Language and Systems*, No. 2, 235-256, April 1987.

[55] Ford, L.R. and D.R. Fulkerson, "Maximal Flow through a Network," *Canadian J. Math.*, Vol. 8, 399-404, 1956.

[56] Ford, L.R. and D.R. Fulkerson, *Flows in Networks,* Princeton University Press, Princeton, N.J., 1962.

[57] Fujiwara, H. and T. Inoue, "Optimal Test Granularity of Test Generation in a Distributed System," *Proc. Intl. Conf. Computer Aided Design*, Nov. 1989.

[58] Fujiwara, H. and T. Inoue, "Optimal Granularity of Test Generation in a Distributed System," *IEEE Trans. Computer Aided Design*, 885-892, Aug. 1990.

[59] Furuichi, M., K. Taki and N. Ichiyoshi, "A Multi-level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-Psi," *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 50-59, 1990.

[60] Galivanche, R. and S.M. Reddy, "A Parallel PLA Minimization Program," *Proc. Design Automation Conference*, 600-607, 1987.

[61] Gallagher, P. Humblet and P.A. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. Programming Languages and Systems*, Vol. 5, 66-77, 1983.

[62] Ghosh, S., "NODIFS: A Novel, Distributed Circuit Partitioning Based Algorithm for Fault Simulation of Combinational and Sequential Digital Designs on Loosely Coupled Parallel Processors," *Technical Report*, LEMS, Div. of Engg., Brown Univ., RI, 1991.

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

[64] Goldberg, A.V., and R.E. Tarjan, "A New Approach to the Maximum Flow Problem," *J. ACM*, Vol. 35, 921-940, 1988.

[65] Grama, A.Y. and V. Kumar, "Parallel Processing of Discrete Optimization Prob-

lems: A Survey," *Technical Report*, University of Minnesota, Minneapolis, MN, 1992.

[66] Grama, A.Y., V. Kumar and V. Nageshwara Rao, "Experimental Evaluation of Load Balancing Techniques for the Hypercube," *Proc. Parallel Computing 91 Conference*, 1991.

[67] Gregoretti, F. and Z. Segall, "Analysis and Evaluation of VLSI Design Rule Checking Implementation in a Multiprocessor," *Proc. Intl. Conf. Parallel Processing*, 7-14, August 1984.

[68] Guibas, L.J., H.T. Kung and C.D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Conf. on Very Large Scale Integration*, California Institute of Technology, 509-525, 1979.

[69] Hachtel, G.D. and P.H. Moceyunas, "Parallel Algorithms for Boolean Tautology Checking," *Proc. Intl. Conf. Computer-Aided Design*, 422-425, Nov. 1987.

[70] Howard, J.K., L. Malm and L.M. Warren, "Introduction to the IBM Los Gatos Logic Simulation Machine," *Proc. IEEE Intl. Conf. Computer Design: VLSI in Computers*, 580-583, Oct. 1983.

[71] Hsueh, M.-Y., "Symbolic Layout and Compaction of Integrated Circuits," *Ph.D Thesis*, University of California, Berkeley, CA, 1979.

[72] Huang, S.S., H. Liu and V. Vishwanathan, "A Sub-Linear Parallel Algorithm for Some Dynamic Programming Problems," *Proc. Intl. Conf. on Parallel Processing*, 261-264, 1990.

[73] Humblet, P.A., "A Distributed Algorithm for Minimum-Weight Directed Spanning Trees," *IEEE Trans. Communications*, COM-34, 345-347, 1983.

[74] Iosupovici, A., C. King and M.A. Breuer, "A Module Interchange Placement Machine," *proc. 20th Design Automation Conf.*, 171-174, 1983.

[75] Ishiura, N., M. Ito and S. Yajima, "High-Speed Fault Simulation Using a Vector Processor," *Proc. Intl. Conf. Computer-Aided Design (ICCAD-87)*, 10-13, 1987.

[76] Ishiura, N., H. Yasuura and S. Yajima, "High-Speed Logic Simulation on Vector Processors," *IEEE Trans. Computer-Aided Design*, CAD-6, 3:305-321, May 1987.

[77] Jacob, G.K., A.R. Newton and D.E. Pederson, "An Empirical Analysis of Performance of a Multiprocessor-Based Circuit Simulator," *Proc. Design Automation Conf.*, 588-593, 1986.

[78] Jayaraman, R. and R. Rutenbar, "Floorplanning by Annealing on a Hypercube Multiprocessor," *Proceedings IEEE International Conference on Computer-Aided Design*, ICCAD-87, 346-349, 1987.

[79] Johnson, T.A. and D.J. Zukowski, "Waveform Relaxation Based Circuit Simulation on the Victor (V256) Parallel Processor," *Technical Report*, IBM, Jun. 1991.

[80] Karypis, G. and V. Kumar, "Unstructured Tree Search on SIMD Parallel Computers," *Technical Report 92-21*, Computer Science Department, University of Minnesota, 1992.

[81] Karypis, G. and V. Kumar, "Efficient Parallel Formulations for Some Dynamic Programming Algorithms," *Technical Report 92-59*, Computer Science Department, University of Minnesota, October 1992.

[82] Kirkpatrick, S., C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, 291-307, May 1983.

[83] Kling, R.M. and P. Banerjee, "Concurrent ESP: A Placement Algorithm for Execu-

tion on Distributed Processors," *Proceedings IEEE International Conference on Computer-Aided Design*, ICCAD-87, 354-357, 1987.

[84] 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.

[85] Kravitz, S.A., R. Bryant and R.A. Rutenbar, "Massively Parallel Switch-Level Simulation: A Feasibility Study," *IEEE Trans. Computer-Aided Design*, Jul. 1991.

[86] Kravitz, S.A. and R.A. Rutenbar, "Multiprocessor-Based Placement by Simulated Annealing," *Proceedings of the 23rd Design Automation Conference,* ACM/IEEE, 567-573, 1986.

[87] Kruskal, J.B., "On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem," *Proc. Am. Math. Soc.*, Vol. 7, 48-50, 1956.

[88] Kumar, C.P.R. and L.M. Patnaik, "Parallel Placement Based on Simulated Annealing," *Proceedings of the International Conference on Computer Design,* xxxxx, 1987.

[89] Kumar, C.P.R. and S. Sastry, "Parallel Placement on Reduced Array Architecture," *Proceedings of the 25th Design Automation Conference,* ACM/IEEE, 121-127, 1988.

[90] Kumar, V., A.Y. Grama and V. Nageshwara Rao, "Scalable Load Balancing Techniques for Parallel Computers," *Technical Report 91-55*, Computer Science Department, University of Minnesota, 1991.

[91] Kumar, V. and V. Nageshwara Rao, "Parallel Depth-First Search, Part II: Analysis," *Intl. J. of Parallel Programming*, 6:501-519, 1987.

[92] Kung, C.P. and C.S. Lin, "Parallel Sequence Fault Simulation for Synchronous Sequential Circuits," *Proc. European Design Automation Conf.*, 434-438, 1992.

[93] Lakshmanan, K.B. and K. Thulasiraman, "On the Use of Synchronizers for Asynchronous Communication Networks," *Proc. Allerton Conf. on Communication, Control and Computer*, Univ. of Illinois, Urbana, Sept. 1987.

[94] 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. SE-15, No. 5, 639-644, May 1989.

[95] Lee, J., E. Shragowitz and S. Sahni, "A Hypercube Algorithm for the 0/1 Knapsack Problem," *Proc. Intl. Conf. on Parallel Processing*, 699-706, 1987.

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

[97] Levendel, Y.H., P.R. Menon and S.H. Patel, "Parallel Fault Simulation Using Distributed Processing," *The Bell Sys. Tech. J.*, 3107-3137, Dec. 1983.

[98] Levitin, S., "MACE: A Multiprocessing Approach to Circuit Extraction," *Technical Report*, MIT, 1986.

[99] Lim, C.F., P. Banerjee, K. De and S. Muroga, "A Shared Memory Parallel Algorithm for Logic Synthesis," *Proc. 6th Intl. Conf. VLSI Design*, Jan. 1993.

[100] Lubachevsky, B.D., "Efficient Distributed Event-Driven Simulation of Multiple Loop Networks," *Comm. of ACM*, 111-123, Jan. 1989.

[101] Mahanti, A. and C. Daniels, "SIMD Parallel Heuristic Search," *Artificial Intelligence*, 1992.

[102] Malhotra, V.M., M. Pramodh Kumar and S.N. Maheswari, "An O(v^3) Algorithm for Maximum Flows in Networks," *Information Proc. Letters*, Vol. 7, 277-278, 1978.

[103] Markas, T., M. Royals and N. Kanopoulos, "On Distributed Fault Simulation," *IEEE Computer*, 40-52, Jan. 1990.

[104] Mayor, P., V. Pitchumani and V. Narayanan, "A Parallel Algorithm for Test Generation on the Connection Machine," *Proc. Intl. Test Conf.*, 9, Sep. 1989.

[105] Miller, D., "Exact Distributed Algorithms for Travelling Salesman Problem," *Proc. Workshop on Parallel Computing of Discrete Optimization Problems*, 1991.

[106] Mohan, S. and P. Mazumder, "Wolverines: Standard Cell Placement on a Network of Workstations," *Technical Report*, Univ. of Michigan, Ann Arbor, MI, 1991.

[107] Monien, B., R. Feldmann, P. Mysliwietz and O. Vornberger, "Parallel Game Tree Search by Dynamic Tree Decomposition," *Parallel Algorithms for Machine Intelligence and Vision*, Eds. Kumar, V., P.S. Gopalakrishnan and L. Kanal, Springer-Verlag, NY, 1990.

[108] Monien, B. and O. Vornberger, "Parallel Processing of Combinatorial Search Trees," *Proc. Intl. Workshop on Parallel Algorithms and Architectures*, 1987.

[109] Moore, E.F., "The Shortest Path through a Maze," *Proc. Intl. Symp. Theory of Switching, Part II*, Univ. Press, Cambridge, MA, 285-292, 1957.

[110] Motohara, A., K. Nishimura, H. Fujiwara and I. Shirakawa, "A Parallel Scheme for Test Pattern Generation," *Proc. Intl. Conf. Computer-Aided Design*, 156-159, 1986.

[111] Mueller-Thuns, R.B., D.G. Saab, R.F. Damiano and J.A. Abraham, "Portable Par-

allel Logic and Fault Simulation," *Proc. Intl. Conf. Computer-Aided Design*, 506-509, 1989.

[112] Nagashima, S., T. Nakagawa, S. Miyamoto and K. Omota, "Hardware Implementation of Velvet on the Hitachi S-810 Supercomputer," *Proc. Intl. Conf. Computer-Aided Design (ICCAD-86)*, Nov. 1986.

[113] Nageshwara Rao, V. and V. Kumar, "Parallel Depth-First Search, Part I: Implementation," *Intl. J. of Parallel Programming*, 6:479-499, 1987.

[114] Nair, R., S.J. Hong, S. Liles and R. Villani, "Global Wiring on a Wire Routing Machine," *Proc. 19th Design Automation Conference*, 224-231, 1982.

[115] Nakata, T., N. Tanabe, H. Onozuka, T. Kurobe and N. Koike, "A Multiprocessor System for Modular Circuit Simulation," *Proc. Intl. Conf. Computer-Aided Design (ICCAD-87)*, Nov. 1987.

[116] Narayanan, V., and V. Pitchumani, "A Parallel Algorithm for Fault Simulation on the Connection Machine," *Proc. Intl. Test Conf.*, 89-93, 1988.

[117] Narayanan, V., and V. Pitchumani, "A Massively Parallel Algorithm for Fault Simulation on the Connection Machine," *Proc. Design Automation Conference*, 734-737, 1989.

[118] Natarajan, K. and S. Kirkpatrick, "Evaluation of Parallel Placement by Simulated Annealing: Part I - the Decomposition Approach," *IBM Technical Report*, Nov. 1989.

[119] Nelson, J.F., "Deductive Fault Simulation on Hypercube Multiprocessors," *Proc. AT&T Conf. Electronic Testing*, Oct. 1987.

[120] Ostapko, D., Z. Barzilai and G.M. Silberman, "Fast Fault Simulation in a Parallel

Processing Environment," *Proc. Intl. Test Conf.*, Oct. 1987.

[121] Ozguner, F., C. Aykanat and O. Khalid, "Logic Fault Simulation on a Vector Hypercube Multiprocessor," *Proc. Intl. Conf. on Hypercube and Concurrent Computers and Applications*, 1108-1116, Jan. 1988.

[122] Ozguner, F. and R. Daoud, "Vectorized Fault Simulation on the Cray X-MP Supercomputer," *Proc. Intl. Conf. Computer-Aided Design*, Nov. 1988.

[123] Pardalos, P.M. and J. Crouse, "A Parallel Algorithm for the Quadratic Assignment Problem," *Proc. Supercomputing 1989 Conference*, 351-360, 1989.

[124] Pardalos, P.M. and G.P. Rodgers, "Parallel Branch and Bound Algorithms for Unconstrainted Quadratic Zero-One Programming," *Impacts of Recent Computer Advances on Operations Research*, Eds. Sharda, R. et al., North Holland, 131-143, 1989.

[125] Pardalos, P.M. and G.P. Rodgers, "Parallel Branch and Bound Algorithms for Quadratic Zero-One Programming on a Hypercube Architecture," *Annals of Operations Research*, 22:271-292, 1990.

[126] Patil, S., "Parallel Algorithms for Test Generation and Fault Simulation," *Technical Report*, Univ. of Illinois, Coordinated Sciences Lab., Sep. 1990.

[127] Patil, S. and P. Banerjee, "A Parallel Branch and Bound Algorithm for Test Generation," *IEEE Trans. on Computer Aided Design*, Vol. 9, No. 3, March 1990.

[128] Patil, S. and P. Banerjee, "Performance Trade-offs in a Parallel Test Generation Fault Simulation Environment," *IEEE Trans. Computer-Aided Design*, 1542-1558, Dec. 1991.

[129] Patil, S., P. Banerjee and J. Patel, "Parallel Test Generation for Sequential Circuits

on General Purpose Multiprocessors," *Proc. Design Automation Conf.*, Jun. 1991.

[130] Patil, S., P. Banerjee and C. Polychronopolous, "Efficient Circuit Partitioning Algorithms for Parallel Logic Simulation," *Proc. Supercomputing Conf.*,361-364, Nov. 1989.

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

[132] Pfister, G., "The Yorktown Simulation Engine: Introduction," *Proc. Design Automation Conf.*, 51-54, Jun. 1982.

[133] Powley, C., C. Ferguson and R. Korf, "Parallel Heuristic Search: Two Approaches," *Parallel Algorithms for Machine Intelligence and Vision*, Eds. Kumar, V., P.S. Gopalakrishnan and L. Kanal, Springer-Verlag, NY, 1990.

[134] Powley, C., R. Korf and C. Ferguson, "IDA* on the Connection Machine," *Artificial Intelligence*, 1992.

[135] Prim, R.C., "Shortest Connection Networks and Some Generalizations," Bell Sys. Tech. J., Vol. 36, 1389-1401, 1957.

[136] Raghavan, R., J.P. Hayes and W.R. Martin, "Logic Simulation on Vector Processors," *Proc. Intl. Conf. Computer-Aided Design (ICCAD-88)*, Nov. 1988.

[137] Ranade, A., "Optimal Speedup for Backtrack Search on a Butterfly Network," *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, 1991.

[138] Rockfeller, R.T., *Network Flows and Monotropic Optimization*, Wiley-Interscience, New York, 1984.

[139] Rose, J.S., "LocusRoute: A Parallel Global Router for Standard Cells," *Proceed-*

ings of the 25th Design Automation Conference, ACM/IEEE, 189-195, 1988.

[140] Rose, J.S., "Parallel Global Routing for Standard Cells," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* 1085-1095, October 1990.

[141] Rose, J.S., W.M. Snelgrove and Z.G. Vranesic, "Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* 387-396, March 1990.

[142] Roucairol, C., "Parallel Branch and Bound on Shared Memory Multiprocessors," *Proc. Workshop on Parallel Computing of Discrete Optimization Problems*, 1991.

[143] Sadayappan, P. and V. Viswanathan, "Circuit Simulation on Shared Memory Multiprocessors," *IEEE Trans. Computers*, 1634-1642, Dec. 1988.

[144] Saletore, V. and L.V. Kale, "Consistent Linear Speedup to a First Solution in Parallel State-Space Search," *Proc. 1990 National Conf. on Artificial Intelligence*, 227-233, August 1990.

[145] Segall, A. and L. Shabtay, "Message Delaying Synchronizers," *Proc. Workshop on Data Structures and Algorithms*, Spain, 1991.

[146] Seiler, L., "A Hardware Assisted Design Rule Check Architecture," *Proc. Design Automation Conference*, 232-238, 1982.

[147] Sherwani, N., *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, MA, 1993.

[148] Shu, W. and L.V. Kale, "A Dynamic Scheduling Strategy for the Chare-Kernel System," *Proc. Supercomputing 89*, 389-398, 1989.

[149] Smart, D. and T. Trick, "Increasing Parallelism in Multiprocessor Waveform

Relaxation," *Proc. Intl. Conf. on Computer-Aided Design*, 360-363, 1987.

[150] Smith, S.P., W. Underwood and M.R. Mercer, "An Analysis of Several Approaches to Circuit Partitioning for Parallel Logic Simulation," *Proc. Intl. Conf. on Computer Design (ICCD-87)*, 664-667, 1987.

[151] Soule, L. and T. Blank, "Parallel Logic Simulation on General Purpose Machines," *Proc. Design Automation Conf.*, 166-171, 1988.

[152] Soule, L. and A. Gupta, "Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation," *Proc. Design Automation Conf.*, 81-86, 1989.

[153] Spiers, T.D. and D.A. Edwards, "A High Performance Routing Engine," *Proceedings of the 24th Design Automation Conference*, ACM/IEEE, 793-799, 1987.

[154] Stunkel, C.B., "Linear Optimization Via Message-Based Parallel Processing," *Proceedings of the Internation Conference on Parallel Processing*, Vol. III, 264-271, August 1988.

[155] Sugiyama, Y. and T. Watanabe, "Parallel Processing of Logic Module Placement," *Electronics Letters*, Vol. 20, No. 5, 219-220, 1984.

[156] Sunderam, V.S., "PVM: A Framework for Parallel Distributed Computing," *Concurrency Practice and Experience*, December 1990.

[157] Tan, S.B., et al, "A Fast Signature Simulation Tool for Built-in Self Testing Circuits," *Proc. Design Automation Conf.*, Jun 1987.

[158] Tham, K.Y., "Parallel Processing for CAD Applications," *IEEE Design Test*, 13-17, October 1987.

[159] Thulasiraman, P., "A Distributed Protocol for the Network Primal-Dual Method

and Simulation on a Shared-Memory Multiprocessor," *M.A. Sc. Thesis*, Dept. of Elect. & Comp. Engg., Concordia University, Montreal, Canada, 1991.

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

[161] 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.

[162] 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.

[163] Thulasiraman, K., M.A. Comeau, R.P. Chalasani, A. Das and J.W. Atwood, "On the Design of a Parallel Algorithm for VLSI Layout Compaction," *Proceedings of International Symposium on Circuits and Systems*, ISCAS 90, 352-355, 1990.

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

[165] Tonkin, B., "Circuit Extraction on a Message-Based Multiprocessor," *Proc. Design Automation Conf.*, 260-265, June 1990.

[166] Trotter, J.A. and P. Agrawal, "Circuit Simulation Algorithms on a Distributed Memory Multiprocessor System," *Proc. Intl. Conf. Computer-Aided Design (ICCAD-90)*, 438-441, Nov. 1990.

[167] Ueda, K., T. Komatsubara and T. Hosaka, "A Parallel Processing Approach for Logic Module Placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. CAD-2, No. 1, 39-47, January 1983.

[168] Ulrich, E.G. et al, "High-Speed Concurrent Fault Simulation with Vectors and Scalars," *Proc. Design Automation Conf.*, 709-712, 1983.

[169] Warshawsky, A. and J. Rajski, "Distributed Fault Simulation with Vector Set Partitioning," *Technical Report*, VLSI Design Lab., McGill Univ., Canada, 1991.

[170] Watanabe, T. and Y. Sugiyama, "A New Routing Algorithm and Its Hardware Implementation," *Proceedings of the 23rd Design Automation Conference,* ACM/IEEE, 574-580, 1986.

[171] Webber, D.M. and A. Sangiovanni-Vincentelli, "Circuit Simulation on the Connection Machine," *Proc. Design Automation Conference,* 108-113, Jun. 1987.

[172] White, J., R. Saleh, A. Sangiovanni-Vincentelli and A.R. Newton, "Accelerating Relaxation Algorithms for Circuit Simulation Using Waveform Newton, Iterative Step Size Refinement and Parallel Techniques," *Proc. Intl. Conf. Computer-Aided Design,* Nov. 1985.

[173] Wolf, W.H. and A.E. Dunlop, "Symbolic Layout and Compaction," in "Physical Design Automation of VLSI Systems." Edited by Preas, B. & M. Lorenzetti, 1988.

[174] Won, Y. and S. Sahni, "A Hardware Accelerator for Maze Routing," *Proceedings of the 24th Design Automation Conference,* ACM/IEEE, 800-806, 1987.

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

[176] Yuan, C.P., R. Lucas, P. Chan and R. Dutton, "Parallel Electronic Circuit Simula-

tion on the iPSC System," *Proc. Custom Integrated Circuits Conf.*, 1-4, 1988.

[177] Zargham, M.R., "Parallel Channel Routing," *Proceedings of the 25th Design Automation Conference,* ACM/IEEE, 128-133, 1988.