Solving Layout Compaction and Wire-balancing Problem using Linear Programming on the Monsoon Multiprocessor

Samir Grover

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

September 1995

ABSTRACT

Solving Layout Compaction and Wire-balancing Problem using Linear Programming on the Monsoon Multiprocessor

Samir Grover

The theme of this work is parallel computing for a CAD application. The layout compaction and wire-balancing problem in VLSI physical design can be formulated as a dual transshipment problem (DTP) which can be written as a linear programming problem and solved using the network dual simplex (NDS) algorithm.

The goal of our work is to implement this algorithm using an implicitly parallel functional language called Id on a small shared memory Monsoon dataflow multiprocessor and study the features of this declarative language to determine quantitatively the amount of parallelism exposed in each feature. In the process, we also examine the parallelism in each phase of the application. We observed that Id's functional features (Higher-order functions, tuples, list and array comprehension, etc.) played a major role to extract the parallelism from our codes. The atomic M-structure allows processes to interact freely, hence, contributing to the parallelism. The resultant parallelism amounted to an average of 20 to 25 operations per cycle.

In this thesis, we compare sequential pivoting with concurrent pivoting strategy in the NDS algorithm. From the experiments, we show that the loss of basisity after concurrent pivoting and then, retaining it makes the NDS slower due to the poor performance of the 0-token spanning-tree building method. Moreover, both pivoting strategies show almost the same amount of parallelism.

ACKNOWLEDGEMENTS

The completion of this thesis would not have been possible without the generous cooperation of my advisor, Herbert H. Hum. I am grateful for his guidance in the formulation and development of this research. I want also to thank Mr. Raghu P. Chalasani for the consultations that he offers in our research work.

I would like to thank Prof. Arvind for providing access to Monsoon Multiprocessor.

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

Finally, none of it would have been possible without the endless love and support of my parents. To them I dedicate this work, and I give my greatest thanks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

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. To achieve this performance, developers of CAD tools are beginning to explore *parallel computing* as a means of accelerating VLSI physical design automation applications. Almost all problems which arise in the physical design cycle (e.g., floor-planning, placement, routing, compaction, etc.) can be formulated in terms of graph (network) optimization problems [7]. Parallel algorithms for these graph problems have been designed and implemented using imperative languages (such as C or Fortran) extended with imperative constructs and annotations to exploit the parallelism[1] for the acceleration of CAD applications [15]. However, *parallel programming* using *extended imperative languages* to exploit parallelism is difficult and error-prone due to the fact that the programmer has to be aware of machine hardware (e.g., number of processors, distribution of the data among processors, etc.) to use it efficiently. The general inability of *automatically* detecting

[1] Parallelism is a general term used to characterize a variety of simultaneities occurring in modern computers. The main advantage offered by parallelism is improved speed.

adequate *parallelism* forces the programmer to explicitly specify the parallelism using the language extensions. To simplify parallel programming, new languages are being investigated which allow the programmer to concentrate on the functionality of the program by expressing parallelism implicitly. Moreover, the programmer does not have to know about the machine hardware to program them. One such language is Id[2] (Irvine Dataflow), an implicit parallel functional language. It is designed to run on Monsoon[3].

In this thesis, we study the features of this language to assess quantitatively which of them are of importance when coding algorithms to solve VLSI design automation problems.

1.1 Parallel Programming

To write a parallel algorithm for a parallel machine, there are two known programming styles, namely, explicit and implicit. Explicit parallel programming (EPP) requires a parallel algorithm to explicitly specify how the processors will cooperate in order to solve a specific problem. The compiler generates code for the instructions specified by the programmer. In implicit parallel programming (IPP), the compiler inserts the constructs necessary to run the algorithm on a parallel computer, and thus this style places a majority of the burden of parallelization, i.e., the problem of partitioning the program into parallel parts (processes), and then synchronizing them if any dependency exists, on the compiler. Some may argue the effectiveness of EPP versus IPP style as viewed by the programmer. We now briefly discuss the EPP style using imperative languages on multiple instruction and multiple data (MIMD) systems. This is followed by a discussion on the IPP style using Id on Monsoon.

[2]This language is developed in M.I.T. Laboratory for Computer Science.

[3]Its architecture is designed at M.I.T. and built by Motorola. It is an experimental multiprocessor.

1.1.1 Why MIMD?

In this model, multiple independent processing elements (PEs), linked together by some interconnection network, work together to process a single problem. All PEs operate asynchronously. The basic unit of computation, into which a program is decomposed for parallel execution, is called a task or a process. Parallel algorithm design for a MIMD system requires investigation of all those operations that can be carried out simultaneously. Then, these operations are grouped into multiple processes and each such process can be assigned to a node or a group of nodes in the graph representing a problem. These processes may communicate, i.e., send signals to synchronize or send the required data value, over the network through global shared memory in a shared-memory MIMD system[4] or passing messages in a distributed memory MIMD system[5] . This simple assignment of nodes to the processes makes the MIMD model an appropriate choice for the VLSI physical design problems.

1.1.2 The EPP Style

When programming shared-memory MIMD systems, the programmer uses synchronizing constructs (e.g., semaphores, barriers, etc.) for shared-memory access; annotations (e.g., fork, join) for process scheduling; and complex constructs for distributing data across system memory. In these systems, tasks are distributed to the processors regardless of where the associated data is. All memory references appear same to the application which greatly simplies programming. Similarly, the programmer uses some annotations for passing messages among the processors when programming distributed-memory MIMD systems. The extension of the sequential languages with *explicit* parallel constructs and annotations allows the programmer

[4]It consists of two or more processors sharing a common bus and memory.

[5]It consists of a network of processors and memory that is distributed as local memory to each processor.

to indicate that certain things may be done in *parallel.* However, software development using these extensions is complex in at least two aspects: *program correctness and efficiency. Correctness* requires the execution results be independent of the number and speed of processors running the program. This requirement can be satisfied only if the parallel tasks are independent of each other or properly synchronized, if a dependence exists. Because it is the programmer's responsibility to code in the correct synchronization to coordinate parallel execution, errors on the part of the programmer can result in race conditions such that the same program produces different answers on different runs. To establish correctness, significant complexities are added to the program. The focus on *efficiency*[6] in parallel program design increases the complexity of software development. It is a difficult task for a programmer to identify and schedule parallel activities small enough to utilize the machine effectively but large enough to keep resource-management overheads associated with managing parallelism reasonable.

1.1.3 The IPP Style

Functional programming (FP) provides a convenient basis for the development of the parallel programming languages that balance the division of work between programmer and the compiler in designing parallel programs. Such languages are not based on a sequential model of computation and can exhibit significant amounts of parallelism implicit in the data-dependencies of the program. These languages allow expressing *what is to be done*, without specifying too much of *how it is to be done* because one does not have to over-specify the details of an algorithm using any imperative constructs and annotations. This property makes these languages *declarative.* In such languages, computational abstractions are expressed through *functions.* Thus, a first-order function takes data objects as arguments and produces

[6]Efficient parallel program design requires perfect distribution of processes on processors in a parallel computer. This problem is known as Load-Balancing.

new data objects as results. The method used to produce new objects from the arguments is abstracted by the function. Higher-order functions take data objects as well as other functions as their arguments and produce new data objects. The support of the immutable data structures (e.g., arrays) makes these languages *implicitly* parallel. The reason is that updating an immutable structure requires some degree of copying instead of rewriting it. Thus, processes cannot interfere because they operate on immutable values and interleaving processes cannot cause *indeterminate* behavior. These semantics allow the compiler to extract a significant amount of the parallelism available in a program automatically.

The functions in FP describes a functional dependence of the output values on input values and are represented as dataflow graphs which correspond to mathematical functions. FP languages are compiled to produce dataflow graphs and these graphs are used as a machine language to run on dataflow computers [1]. The Monsoon [12], a dataflow computer, is designed to run Id [9, 6] programs in a *dataflow style*. In this style, the instruction execution is driven only by the availability of its operands and the instructions are executed in parallel if they are independent. Id abstracts the responsibilities of parallelization from the programmer by making them the responsibilities of the compiler. If an algorithm is specified in Id, one does not explicitly encode parallelism, it is implicit in its operational semantics. Moreover, the programmer does not have to be aware of the parallelism in his program and also does not need to know the details of the machine hardware.

For our study, we have chosen the *layout compaction and wire-balancing* (LCWB) problem in VLSI physical design cycle. This problem is formulated as the *dual transshipment problem* (DTP) and solved using the *network dual simplex* (NDS) algorithm. Our interest in this thesis is to implement this algorithm in Id and investigate the inherent parallelism in it which is exploited by Monsoon to speedup the LCWB problem.

1.2 The LCWB Problem

Consider a set of geometric features L representing a layout (see fig. 1.1),

$$L = \{L_1, L_2, L_3, \ldots, L_N\}$$

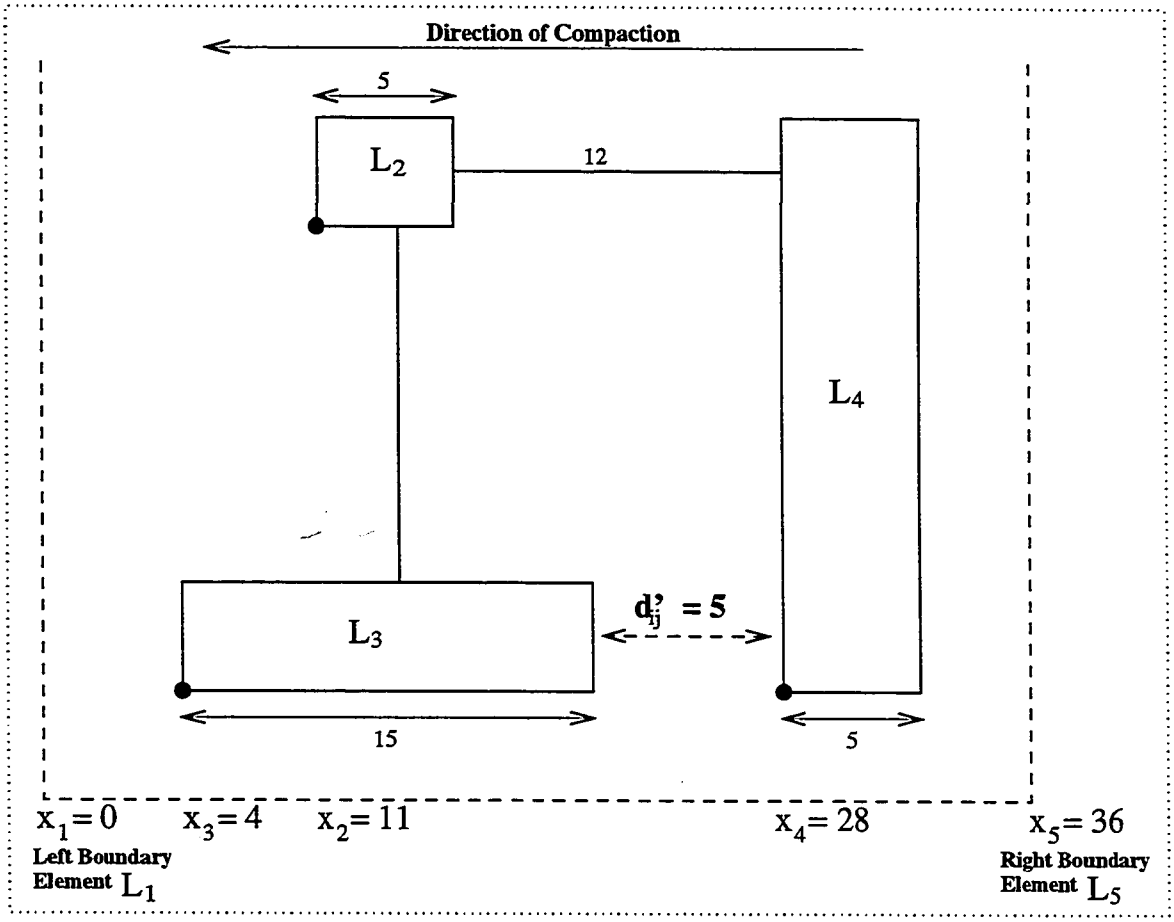

Figure 1.1: Symbolic Layout: a sketch with origins of the elements at their lower-left corners.

where L_1 represents the left boundary, L_N represents the right boundary of the layout and there is a *minimum separation* d'_{ij}, between L_i and L_j, specified by *spacing design rules.* The problem of moving all features as close as possible to one boundary of the layout, such that the overall layout along one dimension is minimized, is called the *compaction problem.* Compacting the layout area can be affected by performing 1-D compaction along the x-axis, and then 1-D compaction

along the y-axis. For clarity, we restrict our discussion to compaction along the x-axis; compaction along the y-axis is similarly performed.

The layout area is minimized by reducing the extra space between features, while conforming to spacing design rules. As a result of this process, the constraining features—they directly affect the compactness of the layout—are placed at their minimal location to find the minimum width of the layout. At the same time, non-constraining features may be placed at one side of the layout creating unnecessarily long wires. Therefore, an optimization (wire-balancing) step is performed to minimize the overall wire length by moving the features against the direction of compaction after a compaction step. These two steps form the layout compaction and wire-balancing (LCWB) problem.

The LCWB problem can be formulated as a DTP [18] and represented by a weighted graph G = (V,E) as shown in fig. 1.2. Each node, $v_i \in V$, represents a feature in the layout and each directed edge $e_{ij} \in E$, pointing from v_i to v_j, represents the minimum spacing requirement[7], d_{ij} between L_i and L_j. Moreover, each feature has a weight, w_i, which can be positively weighted, meaning that moving the component to the right boundary will decrease the total wire length, or negatively weighted, meaning that moving the component to the left boundary will decrease the overall wire length. If it is zero-weighted, it implies that moving the feature left or right would not affect the wire length. Lastly, a position of a feature is denoted by a dual variable (*firing number*), x_i.

The wire-balancing problem can be formulated as a dual linear program when graph G is represented by its adjacency matrix A:

Minimize: WX

subject to:

$$AX \leq D$$

$$X \geq 0$$

[7]This requirement is the sum of left element size and minimum separation d'_{ij}. This is further explained in chapter 2.

where D denotes a minimum spacing weight vector. The inequality, $AX \leq D$, corresponding to each edge e_{ij} of the graph (see fig. 1.2a and 1.2c) is in the form $x_i - x_j + (-d_{ij}) \geq 0$ where $x_i, x_j \in X$ and $-d_{ij} \in D$. Note that the distances are negated to reverse the direction of edges as shown in the formulation at the end of this section. The left-hand side value of the inequality is the *residual token* (it is denoted by r_{ij} in this thesis) of e_{ij} which can be either zero to indicate that both L_i and L_j are at required minimum spacing distance w.r.t. each other or more than zero, say f, to indicate that either L_j (L_i) can be moved to satisfy the minimum spacing requirement by adding (subtracting) f to (from) the firing number of L_j (L_i). These inequalities are the dual constraints in terms of DTP and the minimum spacing constraints in the LCWB problem. The objective function of this dual linear program is to minimize WX. Since the weights are constants, we are left to optimize $x_i \in X$. Since x_N and x_1 denote the positions of the right and left boundary respectively, we also minimize another objective function, $x_N - x_1$, before we minimize WX, and thus solve the compaction problem. In fact, the positions of the features after a compaction step are fed to the wire-balancing step to minimize WX. To solve the compaction problem, the left boundary is assumed to be at zero; thus the position of right boundary is minimized subject to the same constraints expressed as the graph edges.

For example, the LCWB problem for the layout shown in fig. 1.1 is formulated as a DTP as shown below. The generation of input parameters of the DTP from a layout is described in chapter 2.

$$\mathrm{A} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 0 \\ -10 \\ -20 \\ -5 \end{bmatrix}$$

$$W = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \end{bmatrix}$$

where $AX \leq D$

$X \geq 0$

And objective functions to be minimized are:

1) $x_5 - x_1$

2) WX

This formulation is represented as a (*constraint*) graph, G, as shown in fig. 1.2. Note that the thick nodes and edges are artificial, i.e., they are added to solve the problem.

1.3 The Algorithms

Given the LCWB problem as a weighted graph G, the NDS method can be applied to solve it. The first phase in this method is to test the *feasibility* of the input graph G. During this phase, it finds a feasible solution[8] by solving the constraints. As a result, it determines the minimum width of the layout and the corresponding placement of the most constraining elements in one dimension by calculating the critical (longest) path in the layout. For example, the graph in fig. 1.2 becomes the graph in fig. 1.3 after this phase, and its corresponding compacted layout is shown in fig. 1.4. The longest path from all constraining nodes, v_3, v_4 and v_5, to the left boundary node v_1 is shown as dashed edges in the graph (see fig. 1.3) and the corresponding elements in the layout are most constraining (critical) elements. Note that the length of the horizontal wire is increased to 15 from 12 (cf. fig. 1.1).

During the second (*optimization*) phase, it finds an initial basic feasible solution[9]

[8]A feasible solution is a set of values x_i which satisfies the constraints such that each edge $e_{ij} \in E$ has a non-negative residual token.

[9]A solution is called a basic feasible solution of the DTP if graph G has a spanning tree T, such that residual tokens of all edges of T become zero when the features are moved to their locations specified by x_i.. Such a spanning tree, with zero residual tokens on all edges, is called a 0-token

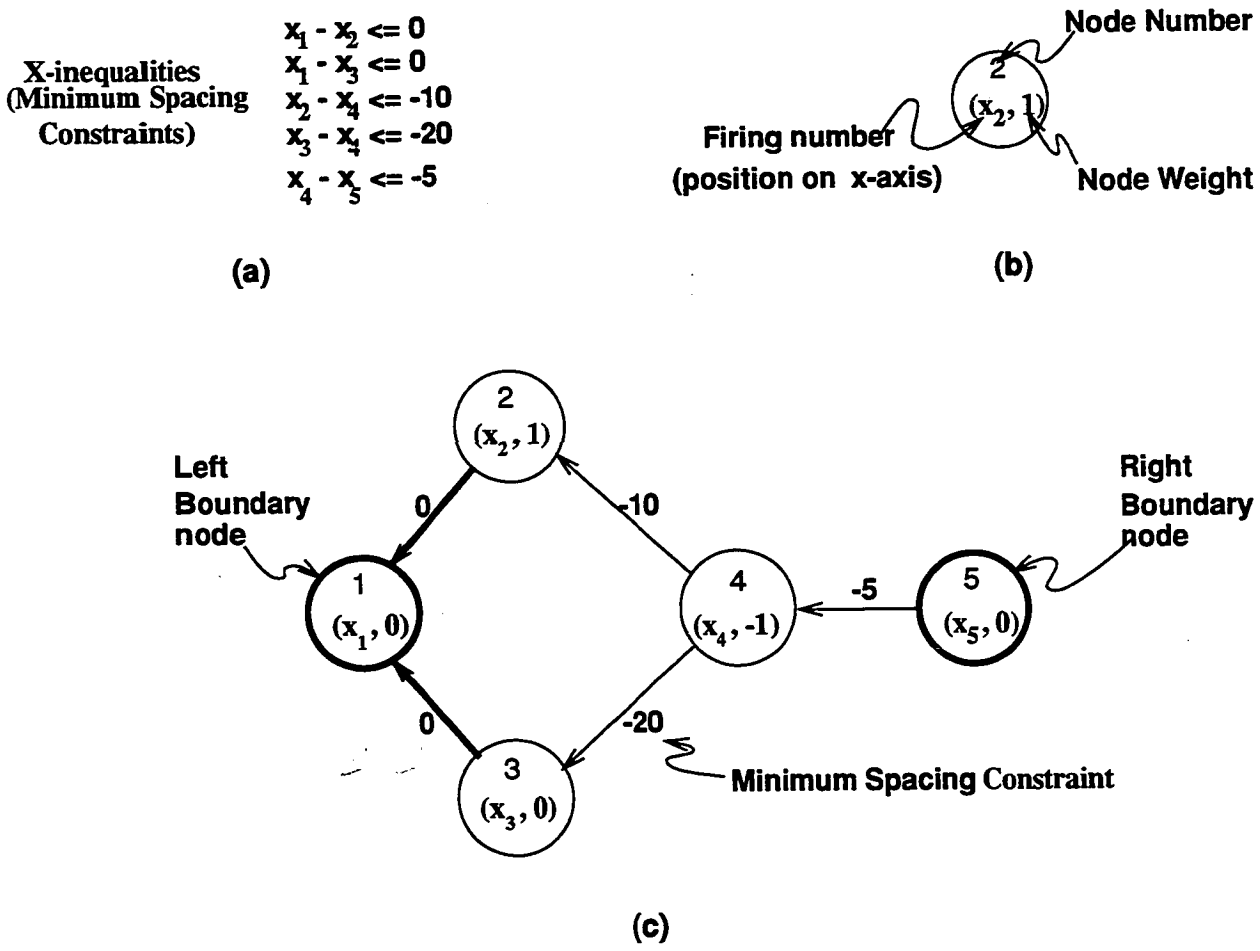

Figure 1.2: X-inequalities derived by scanning the layout and corresponding graph G.

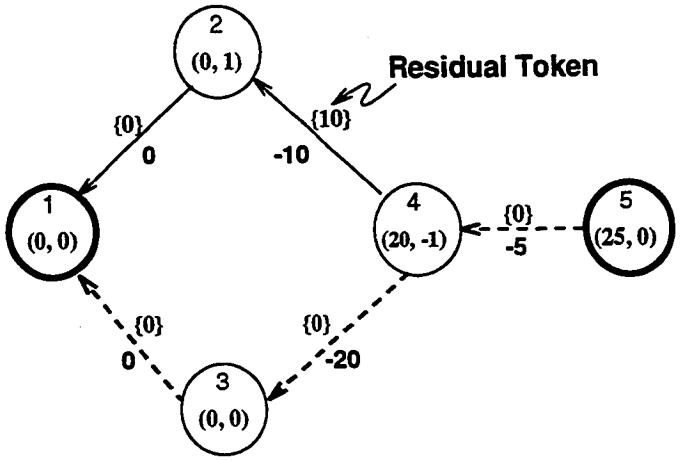

Figure 1.3: Graph G after feasibility testing.

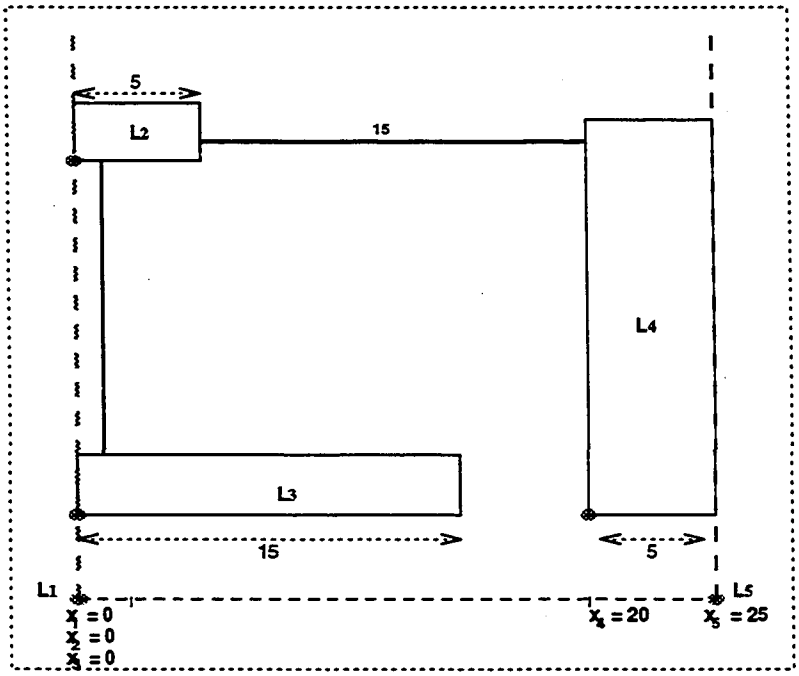

Figure 1.4: Layout after Compaction.

(*bfs*) using shortest-path computations, and then an *optimal bfs* is searched by performing pivoting operations[10]. These operations move some of the elements against the direction of compaction such that the overall wire length is reduced. For example, the graph after the application of this phase is shown in fig. 1.5 and the corresponding compacted and wire balanced layout is shown in fig. 1.6. Note that the non-constraining element, L_2, is moved against the direction of compaction and thus, the horizontal wire length is decreased to 5.

Based on the pivoting steps during an optimization phase, the NDS algorithm can be characterized as a NDS with sequential pivoting (NDS-SP) or a NDS with concurrent pivoting (NDS-CP). The NDS-SP algorithm constructs a *bfs* starting from a feasible solution and moves from one *bfs* to another, performing one pivot step at a time. After each step, this method maintains its *basisity*, i.e., the current

spanning tree (dual feasible tree).

[10]A pivot operation (see section 3.2 in chapter 3) is applied to the zero-edges of the 0-token spanning tree T with one of its adjacent nodes as a leaf node/cluster. This operation moves the feature/features corresponding to that node/cluster of T.

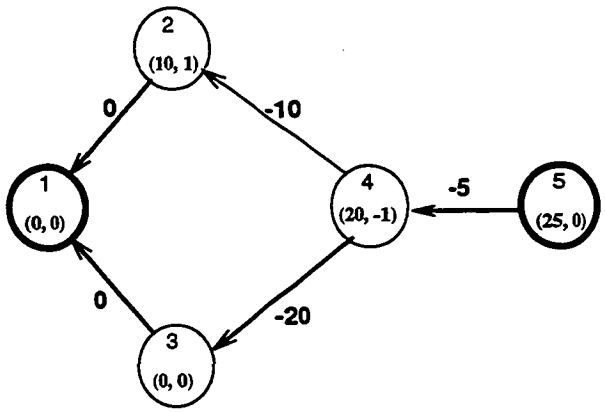

Figure 1.5: Graph G after optimization step.

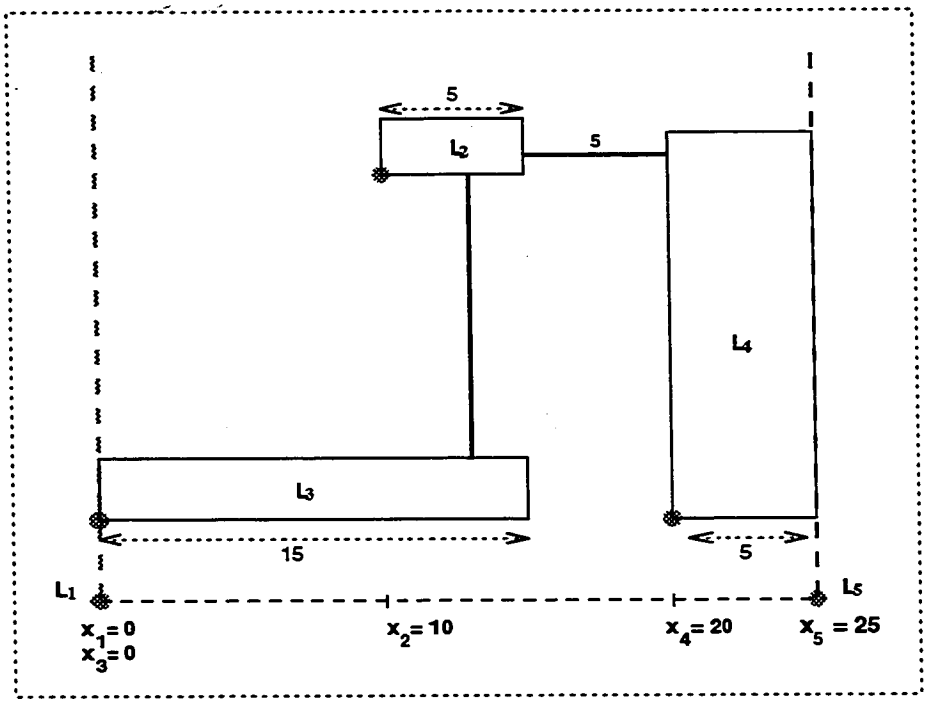

Figure 1.6: Compacted and wire balanced layout.

spanning tree is modified after each firing such that it contains only edges with zero residual token (such edges their $r_{ij} = x_i - x_j + (-d_{ij}) = 0$ are called *zero-edges* in this thesis). The NDS-CP algorithm constructs a *bfs* starting from a feasible solution and performs concurrent pivot steps without destroying *feasibility*. However, it loses basisity of the solution after performing these steps because some edges in the tree become non-zero and the algorithm has to find a *bfs* starting from the current non-*bfs* such that another phase of concurrent pivoting performed.

1.4 The Implementation and Results

We implemented the NDS-CP algorithm in Id to determine its performance on the Monsoon machine. The performance results show that the NDS-CP algorithm, for the relatively small input graph sizes we examined, could not utilize the 1pelis Monsoon machine to its full capacity. For an input graph of 50 nodes and 500 edges, the utilization was only 55%. We investigated the reason for it and found that it had spent most of its time in building the tree than doing other operations. A similar behavior was observed when NDS-CP solved a larger graph of 160 nodes and 3500 edges. This motivated us to re-write the tree-building code in order to improve the overall utilization. In the process, we developed two better approaches to construct the tree, namely, the ZST-SM[11] and ZST-HM[12].

Using the ZST-SM approach, the overall utilization increased to 60% and using the ZST-HM approach, it increased to 68% for the 50-node graph. We run this algorithm using ZST-HM approach on MINT, an instruction-level emulator, to investigate the inherent parallelism in each phase of this algorithm and found that the

[11] In this approach, the original approach which is sequential in nature is divided into two phases. The first phase builds the tree which may have cycles in it. The second phase which is sequential in nature removes these cycles.

[12] In this approach, the second phase of ZST-SM method is parallelized using hierarchical merging to remove cycles.

the hierarchical merging phase of ZST-HM to remove cycles and concurrent pivot steps to find an *optimal bfs* have very little parallelism (about 10 to 16 independent operations) which reduces the average parallelism in the NDS-CP to about 20. This parallelism keeps 1pe1is configuration busy for 68% of the time, and idle for 25% of the time. Among these 25% idles 11% are due to the Monsoon hardware hazard and 14% are due to the lack of work in the algorithm. The remaining 7% are second phase operations. These operations restrict Monsoon to exploit available parallelism because the second phase of the split-phase transaction is executed local to the processor. Normally, the second phase operations are executed on remote processor/memory module containing the required data and a Monsoon processor takes its advantage by exploiting parallelism—executes independent instructions—in order to hide latency and improve performance (see section 4.2 in chapter 4). Moreover, among 68% useful operations 22% are due to RTS which can artificially reduce parallelism by taking a long time[13] to satisfy a frame or heap object request.

From our comparative performance study for the 50 nodes graph, we found that the NDS-SP is faster than the NDS-CP, however, the latter approach executes fewer pivot steps than the former approach. The reason is that the NDS-CP which loses the basisity of solution after performing concurrent pivot steps has to reconstruct a dual feasible tree so that another round of concurrent pivot steps can be performed. However, the NDS-SP maintains the basisity and does not have to build the tree again. The lack of parallelism in the tree-building subalgorithm and concurrent pivot steps in the NDS-CP makes it slower than the NDS-SP for some graphs (see section 5.4 in chapter 5).

The inherent parallelism in the NDS-SP code is measured as 25 on MINT using 50 nodes graph. While coding the algorithm, the parallelism can be expressed in language features such as: single-order and high-order functions; list and array comprehension which are used to construct lists and arrays; conditional statements;

[13]A typical RTS request takes 31 or 32 instructions and it executes these instructions sequentially.

recursive procedures; loops which may be of Doacross type where different iterations sharing a m-structure (introduced later) synchronize using implicit locks, and Doall type where different processes execute independently; parallel blocks where a block consists of single-assignment (variable binding) statements which are executed in parallel if there is no data-dependency between them; and Mutable data-structures (M-structures) for atomic arrays, lists, etc., where each element of the structure carries its own lock for its exclusive update. We found through our parallelism study of the NDS-SP using 50 nodes graph that *list comprehension* contributed the maximum parallelism (35%). The reason for this is that it is used to construct a list of graph nodes (processes) before all loop structures in the code. In this way, we could reduce the overhead of spawning the parallel loop-iterations to minimum. The *for-loops* of Doacross type contributed the second highest parallelism (24%) in our code. This is due to the use the m-structure. Parallel blocks contributed 17% of total parallelism. *Array-comprehension* which are used to construct functional as well as m-arrays and *while-loops* of Doacross type which form the outer loops in our code contributed a similar amount of parallelism (13%). The parallelism due to each construct is further explained in chapter 5.

1.5 Thesis background

Id is a general-purpose parallel programming language. It has three layers (see chapter 4), namely, a pure functional layer, a deterministic layer with I-structures, and a non-deterministic layer with M-structures. Its performance on Monsoon dataflow machine is being studied in the Laboratory for Computer Science (M.I.T.) and elsewhere. One of the major work in this study is presented in [5]. In this paper, they emphasize on the speedup gained by their applications written in Id running on Monsoon multiprocessor. They did not emphasize on the contribution of the Id features to achieve the speedup. In [13], Bohm and Sur have implemented NAS parallel

benchmark Fourier Transform (FT), which numerically solves a three dimensional partial differential equation using forward and inverse fast FTs, in Id to assess which layer of this language is of importance to write scientific codes. To do so, they have implemented this benchmark in three layers of Id and evaluated each layer separately. They found that the M-structure layer allows the largest problem sizes to be run at the cost of about 20% increase in the instruction count, and 75% to 100% increase in the critical-path length[14], compared to the I-structure layer. However, they also did not emphasize on the contribution of the Id features to achieve the performance of their programs.

We have restricted our implementation of NDS[15] algorithm to the pure functional layer and M-structure layer.

1.6 Contributions

The contributions of this thesis are:

- Implementation of the NDS-CP algorithm in *Id*.
- Design and implementation of the **sequential pivot** algorithm for the NDS algorithm.
- A comparative performance study of the NDS algorithm using *sequential pivoting* (NDS-SP) and the NDS algorithm using *concurrent pivoting* (NDS-CP).
- Two designs of 0-token spanning-tree algorithms and the study of their effects on the overall parallelism.

[14]It is the inherent sequential thread in the computations.

[15]In [4], a detail bibliography is given for network dual simplex algorithm and other network optimization algorithms. They have parallelized this algorithm and solved the LCWB problem to show the effectiveness of their algorithm by implementing it on BBN Butterfly multiprocessor using a parallel C language.

- A quantitative study of the parallelism exposed in each Id feature used in our implementation.

1.7 Thesis Outline

The remaining contents of this thesis have been organized in six chapters.

Chapter 2 shows the formulation of a simple layout and the generation of corresponding graph. Chapter 3 discusses the solution of LCWB problem formulated in chapter 2 as a DTP using NDS-SP, and then NDS-CP algorithm. Chapter 4 briefly describes the Monsoon architecture. It also discusses the Id language features which are used to implement our algorithms. Chapter 5 discusses the results of a comparative experimental evaluation of these algorithms. This chapter starts with an evaluation of NDS-CP, and then the effect of tree-building strategies on NDS-CP is studied. This is followed by the parallelism study in the 0-token spanning-tree building strategies on MINT. After selecting the best tree-building subalgorithm, we discuss the parallelism in the first iteration of NDS-CP on MINT, and then we evaluate the NDS-SP. It is followed by a comparison of both algorithms, NDS-CP and NDS-SP, on 1pelis Monsoon and finally, an examination of the contribution to the parallelism of each Id feature in the NDS-SP algorithm. Finally, in chapter 6, we summarize our work in this thesis and point out certain problems for future study.

Chapter 2

Generating the Constraint Graph for the LCWB problem

Constraint graph compaction consists of two steps:

1. Build the constraint graph to indicate the relative positions and the minimum distances required among elements.

2. Solve the constraint graph to minimize the chip area.

In this chapter, we describe the generation of a constraint graph for an example layout. The solution of the graph is described in the following chapter. The example layout in this chapter will be used throughout the next chapter.

2.1 Constraints

In this layout (fig. 2.1), the component numbers are shown inside each component. Each component L_i has a fixed size S_i (shown by dotted double-headed arrows). The minimum separation, d'_{ij}, between any two features is 5 units. A component's position is represented by its left x-coordinate and the actual minimum spacing, d_{ij}, between two components is calculated by adding the left component's width

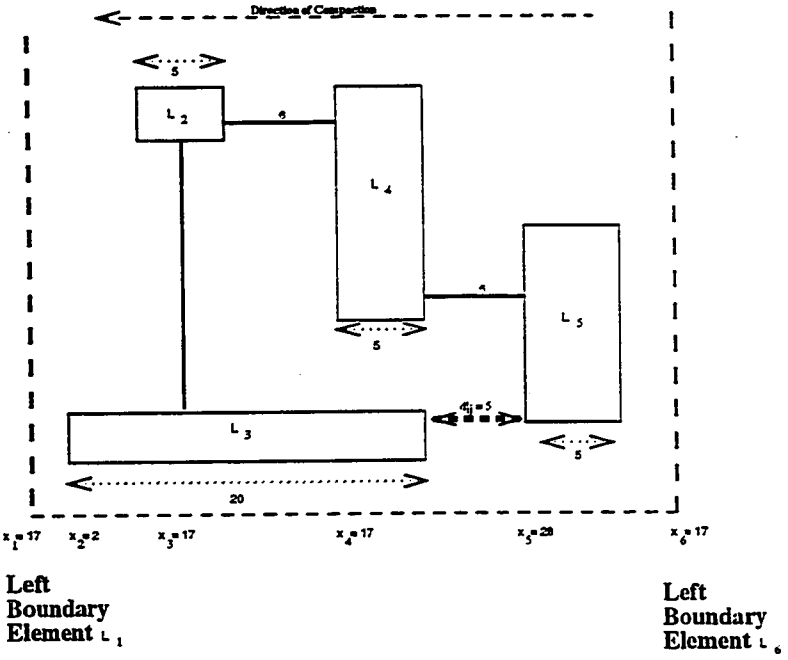

Figure 2.1: Symbolic Layout: A Sketch, with origins of the elements at their lower-left corners

to 5. There is no minimum spacing requirements between a component and chip boundary. Four elements (L_2, L_3, L_4 and L_5) are shown as physically connected to each other and this connectivity information is added to the constraint graph (see section 2.2.2).

The sizes of the features are assumed to be fixed, so the compaction problem is just to move the features closer to reduce the layout area while conforming to minimum spacing rules. Spacing rules can be represented as weighted, directed edges in the constraint graph.

Consider two cells (B and A) as shown in fig. 2.2(a). If they are required to be at least d units apart, then the following minimum spacing linear constraint is represented in the graph as an edge e_{BA} (fig. 2.2(b)):

$x_A - x_B \leq -d_{BA}$

where x_A and x_B represent their x positions. That is, element A is to the left of element B and there is a minimum spacing requirement of d units between them.

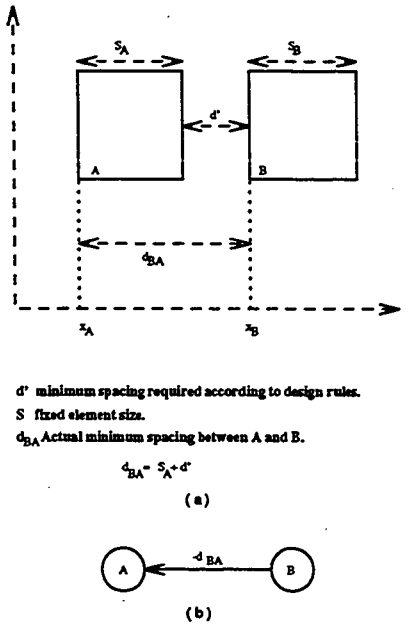

Figure 2.2: Separation constraints and graph representation

2.2 Constraint graph generation

The constraint graph generation[1] step (see fig. 2.3) analyzes the layout to determine spacing rules that must be obeyed, and then adds wire-length minimization information.

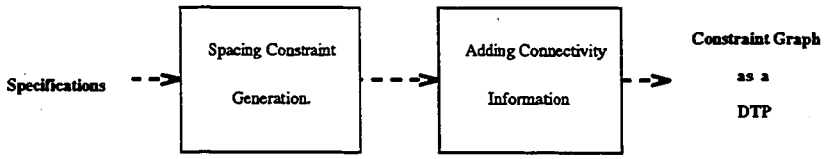

Figure 2.3: Constraint graph generation

[1]In our parallelism study, we are not concerned about building a constraint graph. We assume that the input constraint graph is already generated from the layouts. However, to be complete, we show the generation of a small graph from the simple layout shown in fig. 2.1.

2.2.1 Adding Spacing Information to the Constraint Graph

Scanning this layout shown in fig. 2.1 produces the set of inequalities of the form $x_i - x_j \leq -d_{ji}$ (see fig. 2.4(a)) which is represented as a weighted, directed constraint graph G = (V,E) as shown in fig. 2.4(b). In G, there is a directed edge, $e_{ji} \in E$, between two nodes v_i and v_j, if there is a design rule constraint between the corresponding components. Each inequality corresponds to an edge e_{ji} in the graph with weight $-d_{ji}$. Two special nodes, v_6 and v_1, (corresponding to the right boundary and the left boundary of the layout) are added to the graph. Circuit elements which correspond to nodes with no outgoing edges could be placed at the left boundary, and so edges are added to G directed from each one of these nodes to the left boundary node with weight zero. These artificial nodes and edges are drawn as bold in the graph.

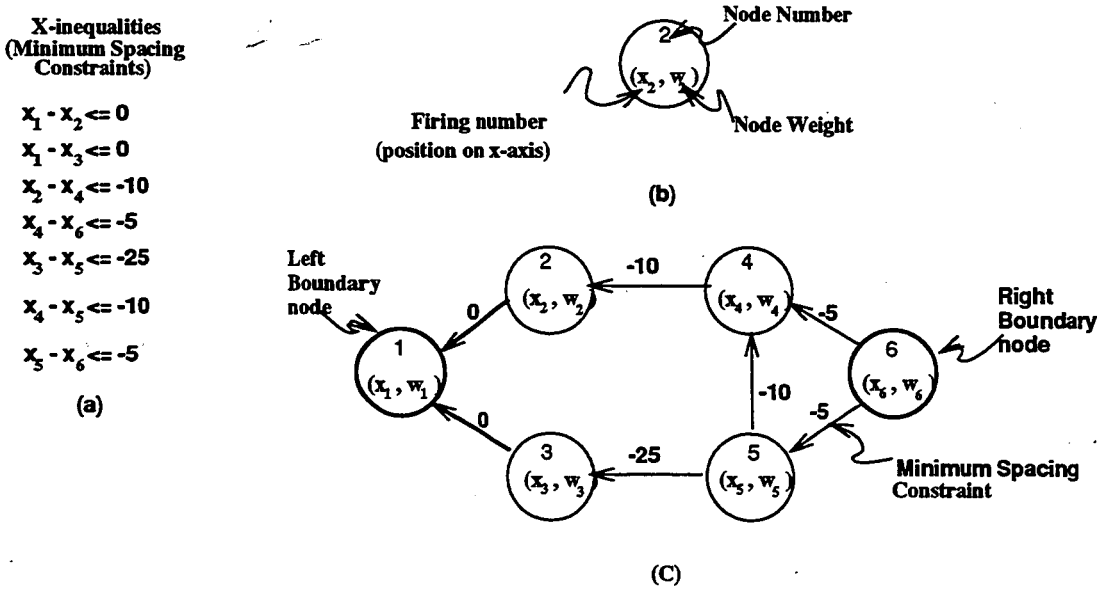

Figure 2.4: X-inequalities derived by scanning the layout and corresponding graph G.

2.2.2 Adding wire-length minimization information to the constraint graph

The greater freedom of motion in constraint-graph compaction makes wire-length minimization compulsory. The compaction in one direction alters only the length of the wires parallel to the direction of compaction and therefore it is sufficient to consider only these wires. The wire-length minimization problem is also transformed to a graph problem in a way similar to the compaction problem by adding pieces of information from the connectivity of the layout to the constraint graph during its construction. This information (denoted by w_i) is added as follows.

In the layout (fig. 2.1), L_2 and L_4 are connected through wire# 1 and L_4 and L_5 are connected through wire# 2. Let us, consider wire# 1. L_2 is assigned weight of 1 (shown in fig. 2.5) to indicate that moving it to the right (opposite to the direction of compaction) will decrease the wire length and L_4 is assigned weight of -1 (fig. 2.5 does not show it) to indicate that moving it to the left will decrease the wire length. Similarly, consider wire# 2. L_4 is assigned weight of 1 (fig. 2.5 does not show it) to indicate that moving it to the right will decrease the wire length and L_5 is assigned weight of -1 to indicate that moving it to the left will decrease the wire length. Thus, $w_4 = 1 - 1 = 0$. All other elements (not connected to the wire), L_1, L_3 and L_6, will have a weight of zero. Note that vertical wires have no relevance in the x-compaction.

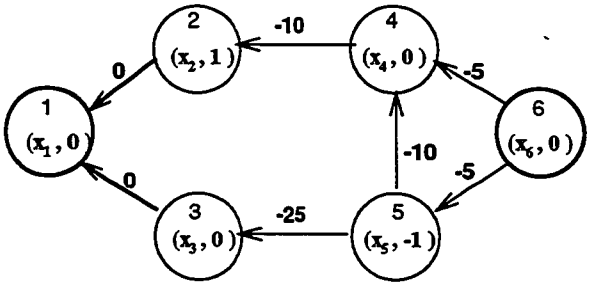

Figure 2.5: the graph G after adding wire-length minimization information

The graph can now be used to compute the new positions for the components

by solving the LCWB problem.

2.3 Summary

This chapter explained the method of generating the constraint graph for the layout shown in fig. 2.1. This LCWB problem can be written as a dual linear program using matrix notation: 7x6 matrix A, positions/firing numbers vector X, minimum spacing vector D, and node weight vector W as shown below and solved using NDS-SP or NDS-CP algorithms.

$$\mathrm{A} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 & -1 \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 0 \\ -10 \\ -25 \\ -10 \\ -5 \\ -5 \end{bmatrix}$$

$$\mathrm{W} = \begin{bmatrix} 0 & 1 & 0 & 0 & -1 & 0 \end{bmatrix}$$

where $AX \leq D$

$X \geq 0$

And objective functions to be minimized are:

1) $x_6 - x_1$

2) WX

Chapter 3

LCWB and the NDS Method

The network dual simplex (NDS) method is based on a dual network flow optimization problem involving a single dual variable per network node. At each iteration, a single node is chosen, and its dual variable or its incident arcs' weights (residual tokens) are changed in an attempt to improve the dual cost. This approach is well-suited for massive parallelization [3] where each node can be assigned to a processor, each adjusting its own dual variable on the basis of local information communicated by adjacent processors.

For our comparative performance study in the next chapter, the LCWB problem is first solved using the NDS-SP (network dual simplex with sequential pivoting) algorithm and then, using the NDS-CP (network dual simplex with concurrent pivoting) algorithm.

3.1 The NDS algorithms

The first step, in both algorithms, is to find a set of values x_i such that each edge $e_{ij} \in E$ has a non-negative residual token. These values form the dual feasible solution of the DTP and are obtained by solving the constraint graph by applying algorithm Feasible and then, an optimization (wire-balancing) step is performed to

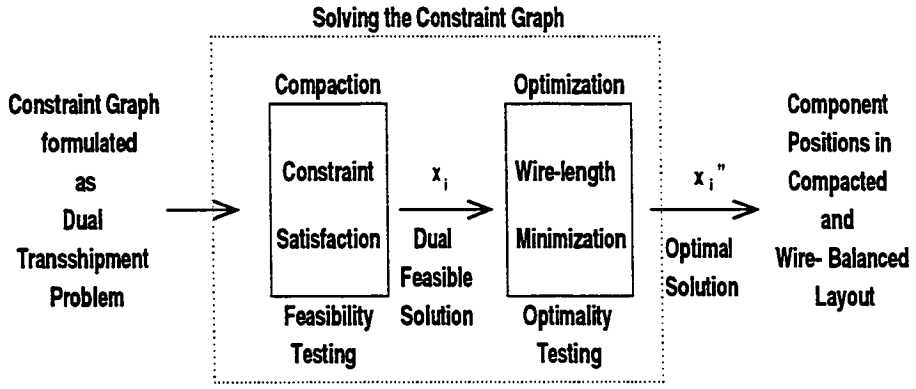

Figure 3.1: Solving a constraint graph

find an optimal basic feasible solution (*bfs*) X. Because algorithm Feasible determines $x_i's$ which satisfy minimum spacing constraints, it finds the dual feasible solution for both problems. Thus, this algorithm serves as a link between the layout compaction and wire-balancing problems.

The optimization phase in the NDS-SP algorithm initializes the *bfs*, and then performs the sequential pivot operations (discussed in section 3.3.4) to find an optimal *bfs* X". This phase terminates when all nodes merge to form a single cluster. The graph at this point is deemed an optimal graph. Whereas, in the NDS-CP algorithm, it is an iterative process which involves the calculations of a *bfs* and concurrent pivot steps (discussed in section 3.4.1) repeatedly. The reason is that at the end of a set of concurrent pivots, the resulting solution may not be *basic*; thus, it has to retain the *bfs* after each concurrent pivoting phase. The graph becomes optimal when all nodes merge to form a single cluster without any firing operations.

Five basic subalgorithms serve as building blocks for the NDS-SP and NDS-CP algorithms, namely, **Feasible** (FT); **Clustering**, **Contraction** and **Shortest-Path Computations & Firing** (C-C-SPC-F); and **BuildTree** (ZST_X). Details and their proof of correctness may be found in [4].

We start with the description of a dual pivot step and then, we explain the basic operations in these algorithms.

3.2 Dual Pivot Step

Let X be the initial *bfs* and T the corresponding dual feasible tree. Consider an edge $e_{ij} \in T$ and let (S,S') be the corresponding fundamental clusters—a cluster consists of one or more nodes—with $v_i \in S$ and $v_j \in S'$ and w_S be the weight of cluster S. The cluster S is represented by a *leaf* node and is connected to the rest of the spanning tree (S') by a single zero-edge e_{ij}. Nodes in S may have non-zero adjacent edges—these edges are not in the current spanning tree T, because their residual tokens are non-zero—connecting them to other nodes in the graph G. A pivot operation (*positive firing operation*) is applied to e_{ij} (an outgoing edge) if $w_S > 0$ where $w_S = \sum w_i$, for all $v_i \in S$.

A dual pivot step performs two types of firing operations. One of them is called a positive firing and its firing condition is mentioned above. For example, consider a 0-token spanning tree T depicted in fig. 3.2 (edges in T are solid lines with weight zero). A zero-edge $e_{3,4}$ connects a positive weighted[1] cluster S represented by leaf node v_3 to the rest of the spanning tree. Thus, $e_{3,4}$ satisfies the firing condition. To perform this firing operation, the firing number of the cluster, f_S, is calculated as follows. Each node, $v_i \in S$, will process its incoming non-zero edges e_{ki} and the minimum residual token of all such edges is assigned to f_S, e.g., $f_S = \min(r_{2,3}, r_{4,5}) = 2$.

The positive firing of a node v_i f_S times, is the operation of adding f_S to its current firing number as well as adding it to the residual token of every outgoing edge $e_{ij} \in E$ and subtracting f_S from the token of every incoming edge $e_{ki} \in E$. Such a situation is depicted in fig. 3.3. All nodes (v_3, v_5 and v_1) in cluster S are fired by two. Their firing numbers are increased by two and residual tokens of incoming and outgoing edges of S are modified such that no resulting residual token becomes

[1]After clustering (only during any pivoting strategy but not while finding an initial bfs) of v_1, v_5 and v_3 having node weights as 1, 0 and 1 respectively, each one is assigned its representative (v_3) weight which becomes 2 ($w_3 = 1 + 0 + 1 = 2$).

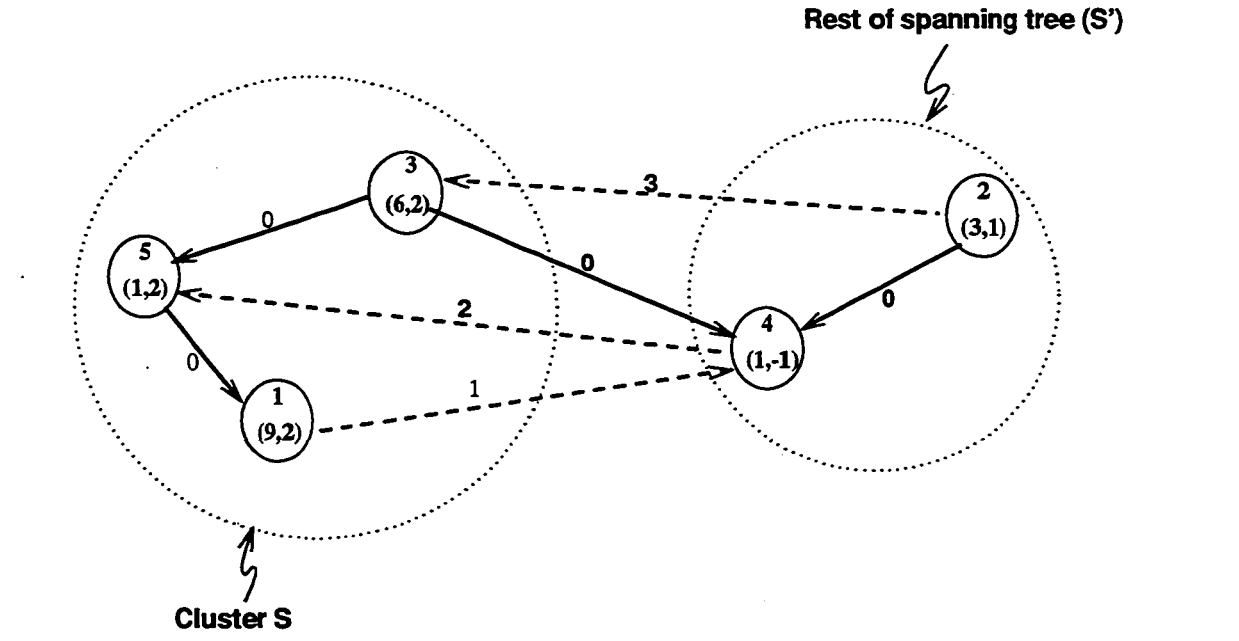

Figure 3.2: Dual pivot step: initial state (before positive firing)

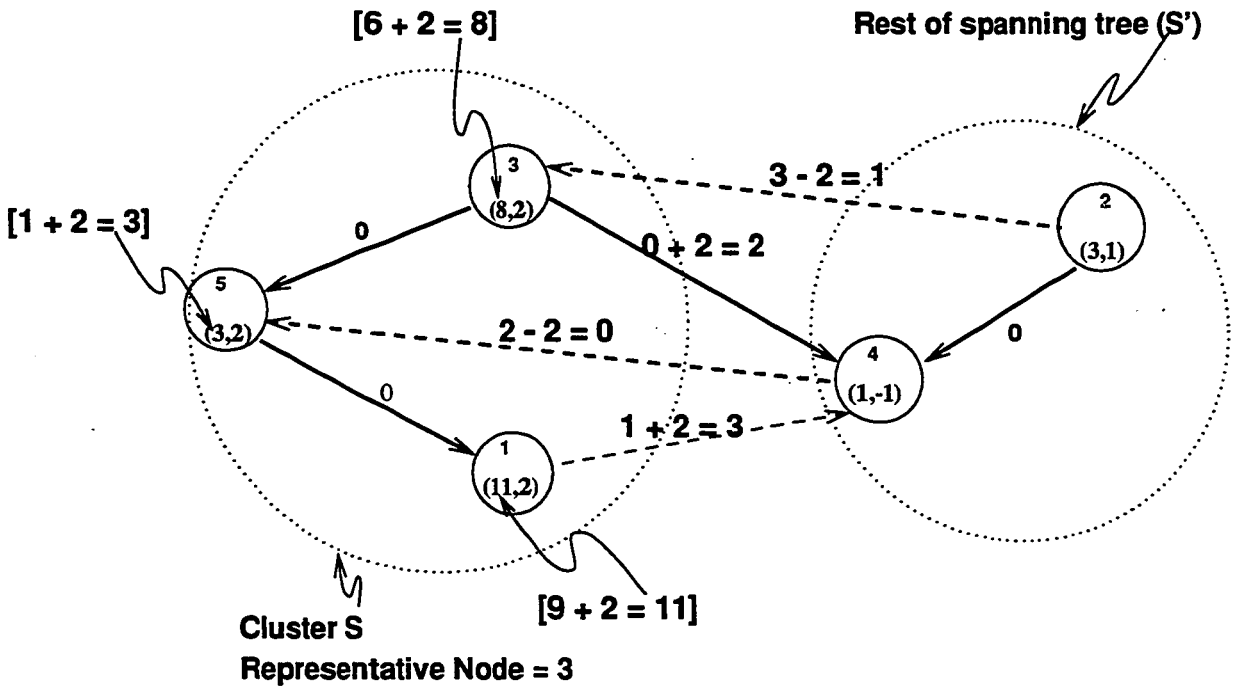

Figure 3.3: Positive firing: calculate f_S and fire

negative.

As a result of this firing, the residual token of zero-edge $e_{3,4}$ becomes two. Thus, this tree T loses its *basisity*, because the residual token of each edge in the tree must be zero.

In the case of sequential pivoting, the tree is modified by deleting the zero-edge e_{ij} (it has become non-zero as a result of firing and it is called a *leaving edge*, e.g., e_{34} in this example) and replacing it by the non-zero edge e_{ki} (it has become zero as a result of firing and is called an *entering edge*, e.g., e_{45} in this example) after any firing. It is done to maintain the basisity of the solution. Such a situation is depicted in fig. 3.4.

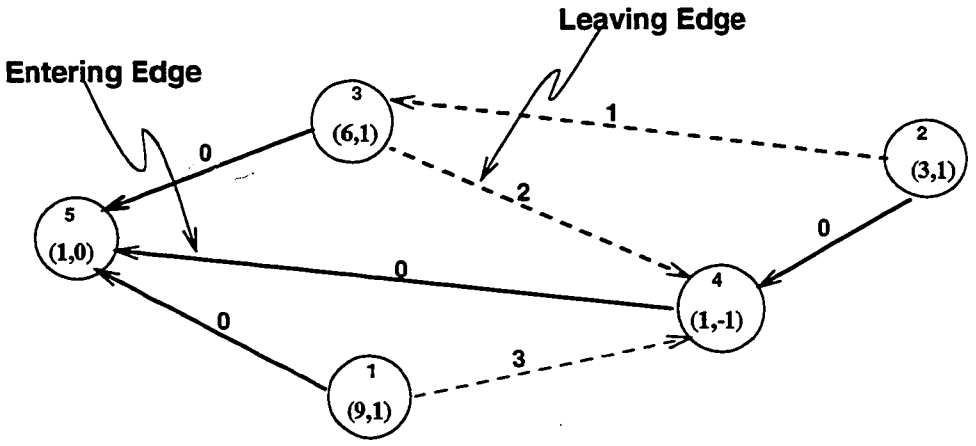

Figure 3.4: Sequential pivoting maintains the basisity

However, the concurrent pivoting method does not modify the tree and continues performing pivot steps on a non-basic tree. This will be explained further in section 3.4.

The second type of firing operation during a pivot step is the negative firing which is the inverse of the positive firing (see [4]). A negative firing operation is applied to an edge e_{ji} (an incoming edge) where $v_i \in S$ and $v_j \in S'$ if $w_S < 0$. In a negative firing, the firing number f_S is the minimum residual token of all the outgoing edges (from each $v_i \in S$ to S') not in the current spanning tree and before

firing S, f_S is negated.

The advantage of employing both forms of *firing* during a pivot step is that after the computation of a fundamental cluster, it is not discarded even if it is negative weighted, which saves a lot of computations. However, as a result of negative firing, firing number of nodes may become negative at the end of a concurrent pivot phase because negative firing basically moves the left-boundary beyond zero. These are corrected by adding the least negative firing number to all the firing numbers. Both forms of firing operations during a pivot step can not be employed concurrently. So, we fire positively first then we fire negatively (see [4]); the ordering is inconsequential.

Finally, firing a node/cluster, v_i, zero times is the *degenerate* pivot [17, 4] operation. These pivot operations degrade the performance of any pivoting strategy. If a degenerate pivot is found, then the current 0-token spanning tree is modified by replacing the corresponding zero-edge, e_{ij}, with another zero-edge, e_{ik}, not in this tree and pivoting process is restarted. Thus, this process changes one basic tree to another without improving the solution and is called *degeneracy.*

3.3 The NDS-SP Algorithm

Given a DTP, the NDS-SP finds a dual feasible solution, X, using subalgorithm Feasible. Then, it constructs a *bfs*, X', such that $WX' \geq WX$ using three subalgorithms, namely, Clustering, Contraction, and Shortest path computation & firing. The process of constructing a *bfs* is an iterative process as shown in fig. 3.5. At the end of this process, it results in a set of values $x'_i \in X'$ such that each edge $e_{ij} \in E$ has a non-negative residual token and zero-edges span all the nodes in the graph. At this point, a 0-token spanning tree T is formed using a subalgorithm ZST-X. Given such a tree, the sequential pivoting (subalgorithm *sPivots*) essentially

involves traversing this tree bottom up (*starting from a leaf*), identifying a fundamental cluster and firing it. During this pivoting strategy, the computations proceed in iterations. Each iteration (dual pivot step) processes a zero-edge of the tree to improve the current *bfs*. This flow of the NDS-SP algorithm is visualized in fig. 3.5. We will explain the process of each step on the graph shown in fig. 2.5.

3.3.1 Feasibility Testing: The Compaction Step

Algorithm Feasible solves the minimum spacing constraints and assigns values to all the dual variables, $x_i \in X$. To do so, it performs two steps as follows.

During the first (initialization) step, each node v_i sets x_i to zero to place all elements at the left boundary of the layout. Then, each node sets x_i to the value of the most negative edge weight, d_{ij}, of its outgoing edges (each node traverses its outgoing edges to calculate the most negative edge weight) to move each element towards the right boundary, and thus places each one at its minimum spacing distance from its nearest left element. The result of this step is visualized in fig. 3.6 where each node v_i is assigned $x_i = Max_j\{\text{-}d_{ij}\}$, for each $e_{ij} \in E$.

During the second (testing) step, each node v_i calculates most negative residual token (say r) of all its outgoing edges and updates its firing number, x_i, to place itself at its longest path distance from the left boundary node v_1. This step works in phases where each phase executes N iterations. During each phase, each node v_i calculates r and moves itself by adding r to its firing number *atomically* because it may interfere with its adjacent node if that is also updating its firing number. In any phase, if a node updates its firing number, then it sets a flag to true to indicate that another phase will be executed. This subalgorithm will terminate when no node modifies its firing number in a phase. The result of this step is visualized in fig. 3.7. In a phase, each node v_i is assigned $x_i = Max_j\{\text{-}r_{ij}\}$ where $r_{ij} = d_{ij} + x_i - x_j$, for each $e_{ij} \in E$.

Solving a system of dual constraints produces a complete placement in one

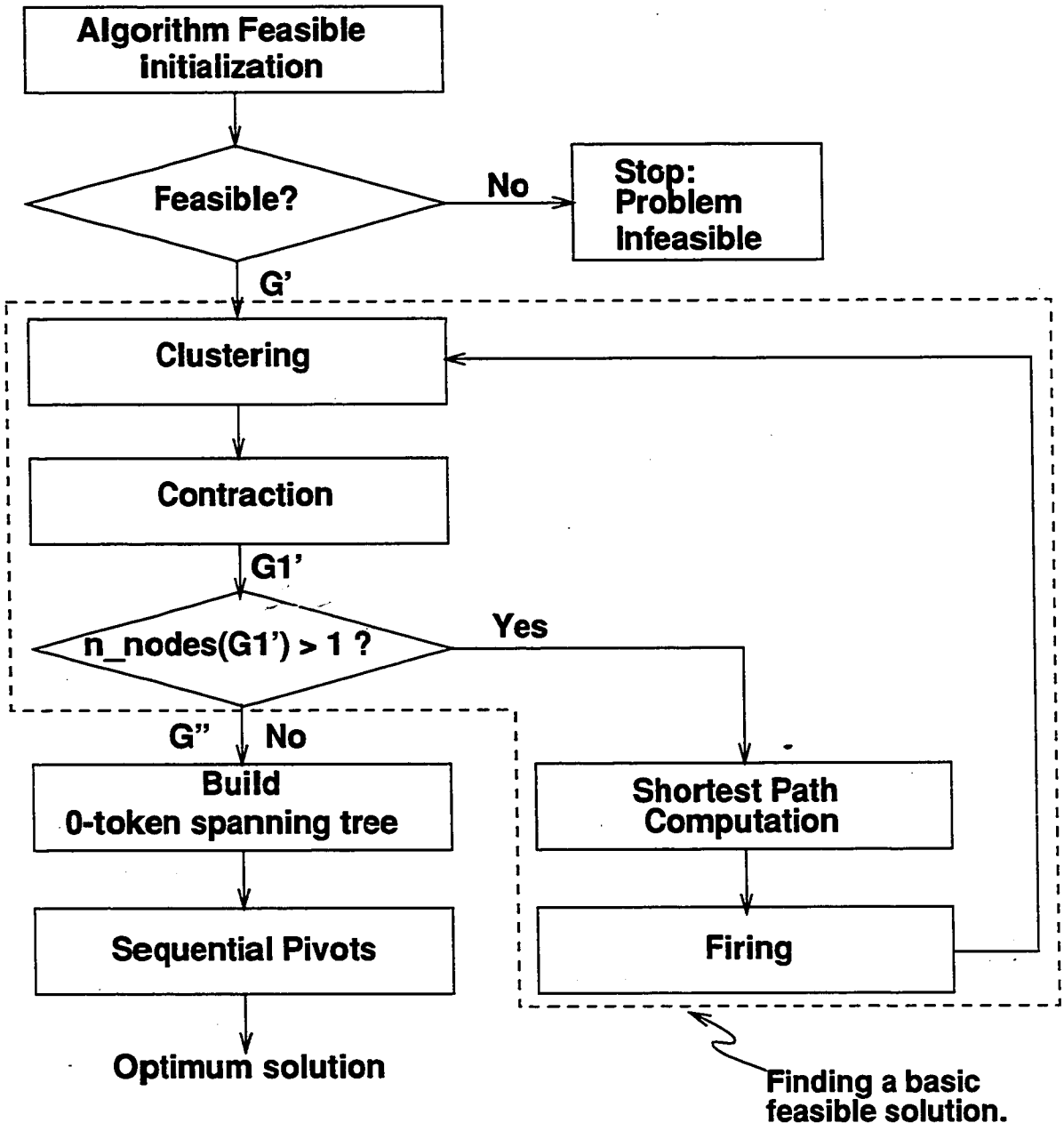

Figure 3.5: Flow diagram of NDS-SP algorithm

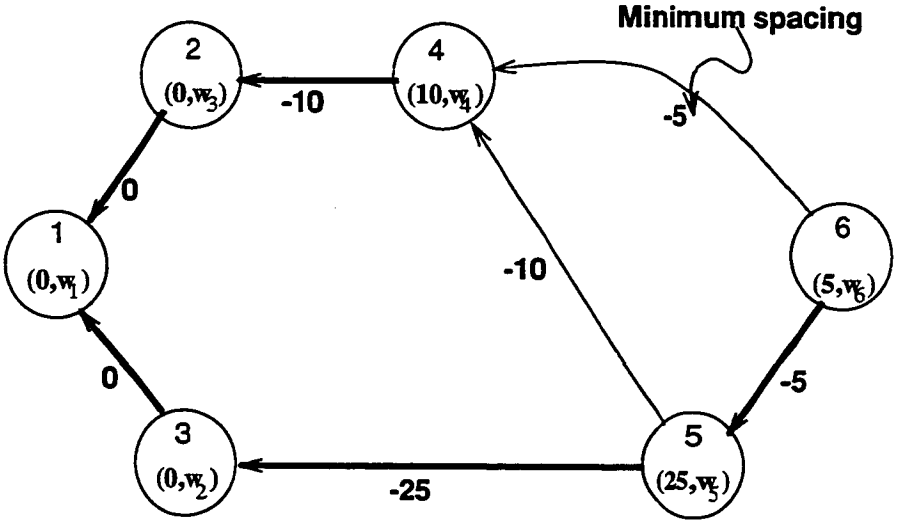

Figure 3.6: Algorithm Feasible: initialization step

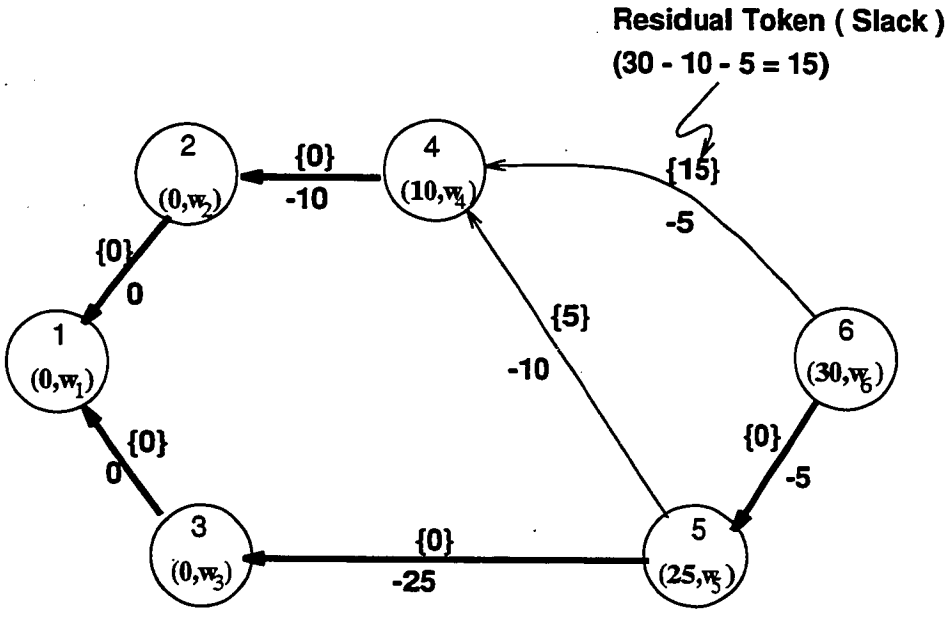

Figure 3.7: Algorithm Feasible: testing step

dimension for the elements in the layout [4]. The x_i's obtained at the end of feasibility testing represents the positions of the different circuit elements. The maximum x_i ($x_6 = 30$), in fact, is the length of the most negative edge weight directed path in graph G from the right boundary node to the left boundary node ($v_6 \rightarrow v_5 \rightarrow v_3 \rightarrow v_1$), which is also the minimum width of the layout. This path is also called the longest path or critical path. As a result of this process, the components representing nodes v_6, v_5, v_3 and v_1 are placed at their required minimum spacing distance. This is indicated by zero-edges along this path. These components are called the constraining (critical) elements (shown in bold in fig. 3.8), and they directly affect the compactness of the layout.

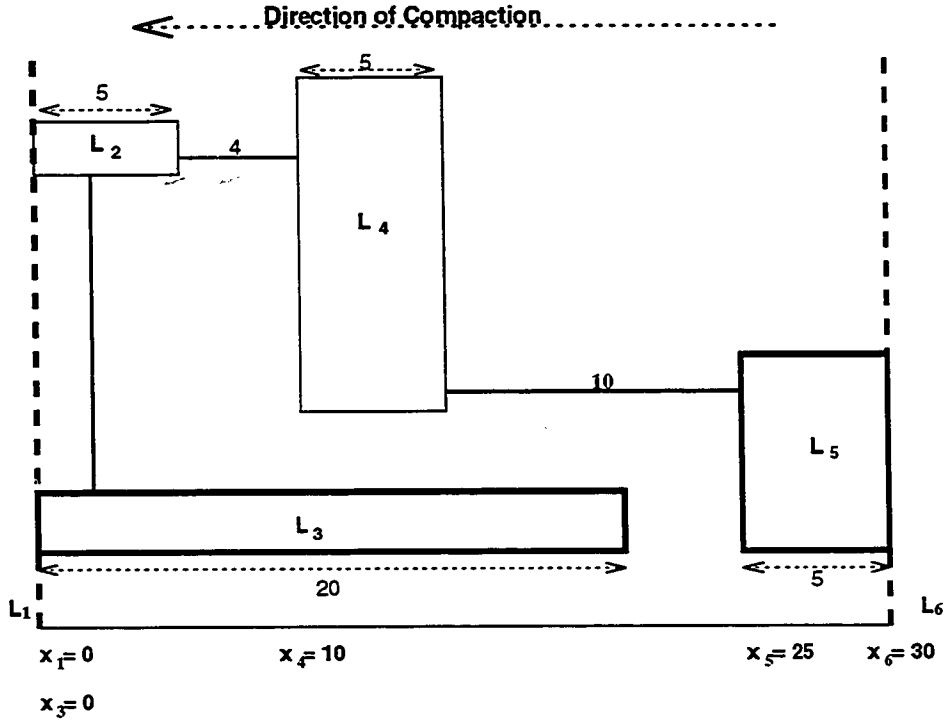

Figure 3.8: Layout-II after Compaction

All elements (nodes) which are not on the longest path are the *non-constraining* elements (L_2 and L_4 in fig. 3.8) and their placement by this subalgorithm increases the overall wire length by two (cf. fig. 2.1). Moreover, L_4 can be moved to the right upto five units without increasing the width of the total layout. This can be

calculated by finding the minimum of the residual tokens of the incoming edges at v_4. The objective function of wire-balancing problem is -25. As we can see from fig. 3.7, it is not minimum (optimum) because L_4 can be moved to reduce the overall wire length, therefore the objective can optimized in the following optimization phase.

Significance of Residual Tokens

Residual tokens play an important role here. Consider two elements L_A and L_B (referring to fig. 2.2 in chapter 2). Their positions are denoted by x_A and x_B. Minimum spacing between these two elements is given by d_{BA}. If element L_A is at x_A then, x_B should be greater than or equal to $d = x_A + d_{BA}$. If $x_B = d$ then both elements are at their required minimum spacing. If $x_B > d$ then element L_B may be moved to the left by $r_{BA} = x_B - (x_A + d_{BA})$ units or L_A can be moved to the right by r_{BA} to minimize the overall wire length.

3.3.2 Constructing an Initial Basic Feasible Solution

In fig. 3.8, some of the residual tokens on the edges are zero. This fact is exploited while finding a *bfs*. The graph at the end of *feasibility testing* is G'. To find a *bfs*, X', node set V is partitioned into subsets (clusters) $S_1, \ldots, S_k$ such that if a cluster, S_i, has a node with negative weight, then all other nodes[2] reachable from this node by a directed path of zero-edges are also in S_i. After this partitioning of V into clusters, only nodes with a positive weight are fired by p which is the *shortest path distance* from the negative weighted clusters to them.

The construction of a *bfs*, X', from a given dual feasible solution, X, involves repeated applications of the following steps.

1. Clustering

2. Contraction

[2]They may be negative, zero or positive weighted nodes.

3. Shortest path computations and cluster firings.

These steps may be repeated until all nodes coalesce into a single cluster after the Clustering step. Now, we will explain the working of each step on the graph shown in fig. 3.7.

Clustering

Clustering involves finding for each non-positive node[3] v_i, the set of all nodes reachable from v_i by directed paths of zero-edges. The least-numbered non-positive weighted node in a cluster is used as the representative of the cluster. In fig. 3.7, v_1 is reachable from v_4 as well as from v_6 by a directed path of zero-edges. Thus, these nodes will form a cluster, say S_1, represented by v_1.

Note that there is only one node v_2 with a positive weight. However, it has adjacent zero-edges to non-positive nodes v_1 and v_4 which indicates that element L_2 is at its minimum spacing distance from L_1 and L_4. Therefore, it is also represented by v_1.

To perform clustering on fig. 3.7, each node v_i is associated with an attribute called a representative (say source[i]). It is initialized to ∞ if the weight of the node is positive, otherwise it is initialized to i. Next, two steps are performed in sequence.

During the first step, each node, v_i, with source[i] $\neq \infty$ starts a process which examines the representative information associated with each node, v_j, of its outgoing zero-edge and performs the following statement.

If source[j] $= \infty$, then source[j] is set to source[i]. Otherwise, find the root[4] of v_j (say rj). If rj $<$ ri where ri is the root of v_i, then source[i] = rj, otherwise source[j] = ri.

[3]It is a node with zero or negative weight.

[4]If source[i] = i then v_i is the root of the cluster containing i. Otherwise, the root is found by traversing the list of representative nodes starting from the current representative until a node is found which is the representative of itself.

This step is executed in phases. In each phase, all active nodes work in synchrony. Each active node which updates its source information resets its flag to false to indicate that it will be inactive in the next phase. This step is repeated until all nodes propagate their representative information to its adjacent nodes.

For fig. 3.7, this step is repeated two times. During the first phase, five processes corresponding to the five nodes, v_1, v_3, v_4, v_5 and v_6, are active. The process corresponding to v_1 does not perform any action because there is no outgoing edge. However, the other four processes perform the above statement. The representative of v_2 becomes 4 after the first phase, so it propagates its representative information to its outgoing edge in the second phase. There is only one active process corresponding to v_2 during the second phase. It modifies its representative, v_4, information to 1 according to the statement mentioned above. The state of the graph, after this step, is shown in fig 3.9.

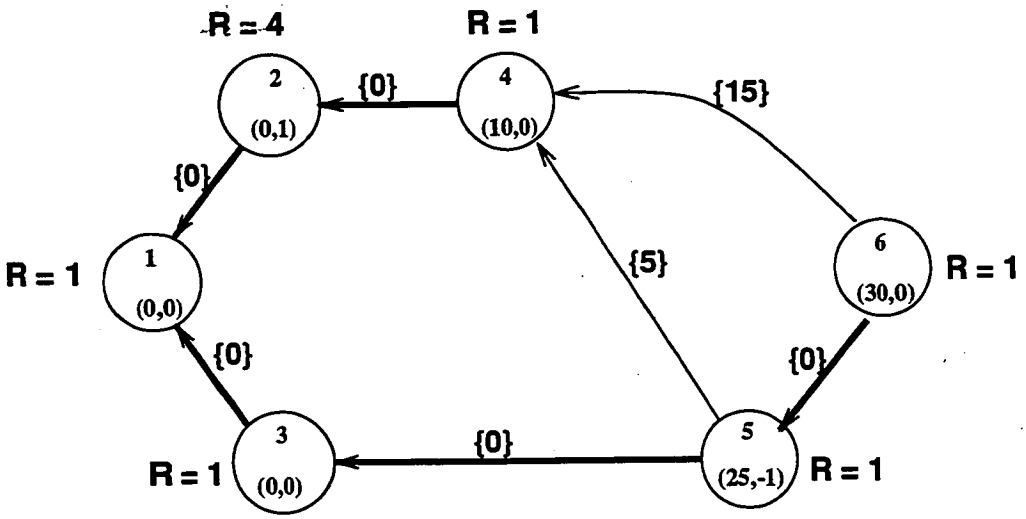

Figure 3.9: Step-I: Clustering of G'

In the second step, all nodes work in synchrony where each node traverses the chain of representatives to find its root (say rt), and then to write source[i] = rt as shown in fig 3.10. Thus, all nodes are represented by v_1 and a cluster (say S_1) is

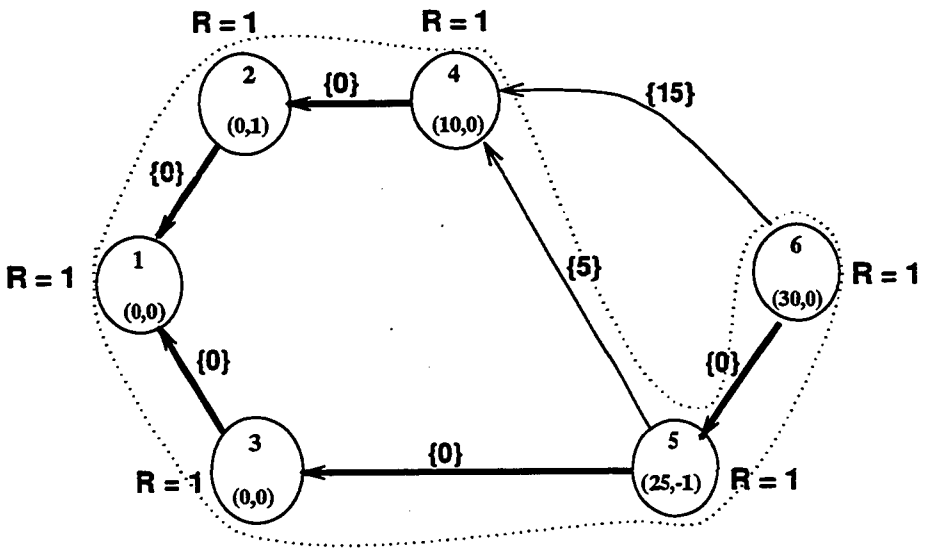

Figure 3.10: Step-II: Clustering of G'

formed.

Contraction

Let $S_1, \ldots, S_k$ be the clusters formed by the clustering step and be represented by their representatives. During this step, all the nodes in each cluster are contracted and a graph G1' having at most one edge directed from one cluster S_i to another S_j is constructed. The residual token of this edge e_{ij} will be the minimum of the residual tokens of all edges in G' directed from a node in S_i to a node in S_j. Thus, the contracted graph G1' will have only nodes representing a cluster in G' at the end of contraction step. The weight of a representative node in G1' will be the sum of weights of all the nodes in the cluster. In this example, because all nodes have the same source (representative), hence, the contracted graph G1' has only one node v_1 and its weight is zero as shown in fig. 3.11.

To construct G1', N processes corresponding to each node in G' become active. Each node $v_i \in G'$ traverses a list (*ActiveOutNodes*) of the nodes on its outgoing edges and performs the following statement:

If the root of any of these nodes (say rj) is same as that of v_i (say ri), then it

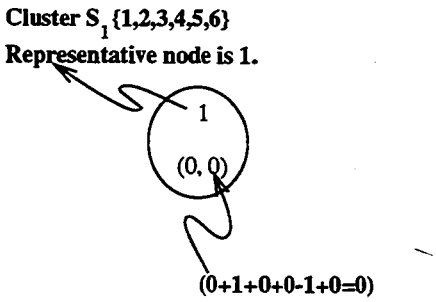

Figure 3.11: Contraction: G1'

removes v_j (i.e., deletes e_{ij}) from the *ActiveOutNodes*. If $rj \neq j$ and $rj \neq ri$ then it deletes e_{ij} and adds a new edge, $e_{i,rj}$. The weight of this new edge will be the same as that of the deleted edge e_{ij}.

After updating its *ActiveOutNodes*, v_i performs the following statement.

If $ri \neq i$ then the weight of ri is increased by the weight of v_i and ActiveOutNodes[ri] is set to ActiveOutNodes[ri] ∪ ActiveOutNodes[i].

All nodes work in synchrony with each other while accessing their root's *ActiveOutNodes* structure. Each node locks its *ActiveOutNodes* so that it will wait for its use in the later statement until it is updated by the former statement which results in *producer-consumer* parallelism during each iteration.

For example in fig 3.10, the process of v_5 is processing its adjacent nodes v_3 and v_4. However, roots of both, v_3 and v_4, are 1 which is also the root of v_5. Hence, both edges, $e_{5,3}$ and $e_{5,4}$, are deleted. Similarly, other edges are also deleted. In this way, all nodes in the cluster are contracted to form a new graph having only one root node, v_1, representing the cluster S_1.

Shortest Path Computation and Firing

Shortest Path Computation (SPC) and Firing (F) operations find for each cluster S_i with positive weight in G1' a shortest path to it from a cluster with non-positive weight. The weight of this path specifies the number of times each node in cluster S_i could be fired without resulting in any negative residual token. In G1', if a cluster

with a positive weight (say S) is not reachable from any cluster with a non-positive weight, then this indicates *unboundedness* of the given DTP, because S can be fired an unbounded number of times increasing the value of WX to an unbounded value.

To perform SPC, each node is assigned its two variables, namely, *distance* and *newDistance*. There are two steps in this subalgorithm. In the first (initialization) step, each node i will initialize its *distance* variable to ∞ and each start-node[5] will initialize its *newDistance* variable to zero while all other nodes will initialize their *newDistance* variables to ∞.

The second step is an iterative one which is repeated until all positive weighted clusters reach their shortest path distances from non-positive clusters. This step is also executed in phases. During each phase, only nodes with $newDistance_i < distance_i$ will be active. Each active node v_i sets $distance_i$ to $newDistance_i$ and updates the $newDistance_j$ variable *atomically* for each of its outgoing nodes as follows.

$$newDistance_j = \text{Min}\{newDistance_j, ResidualToken_{ij} + newDistance_i\}$$

This iterative step terminates when $distance_i$ becomes $newDistance_i$ for each node. Note that all active nodes in a phase work in synchrony. They may synchronize at the *newDistance* variable of their outgoing nodes.

To perform cluster firing (F), the root (say ri) of each positive weighted cluster S_i becomes active and all nodes in that cluster are fired positively by *distance[ri]*, i.e., each one is moved towards the right boundary. In this process, all eligible clusters are fired concurrently.

In our example, SPC-F operations are not needed to move any element with respect to left boundary because each one is at its longest-path length from it. This indicates that the dual feasible solution obtained after *feasibility testing* is essentially a *bfs* as shown in fig. 3.12. The resulting *bfs*, X' = {0, 0, 0, 10, 25, 30}, remains the same and the objective function calculated as $WX' = -25$, is the same as WX.

[5]If the weight of a cluster is non-positive, then the root of this cluster is chosen as a start-node.

The graph at this point is denoted as G". In this graph, zero-edges span all the

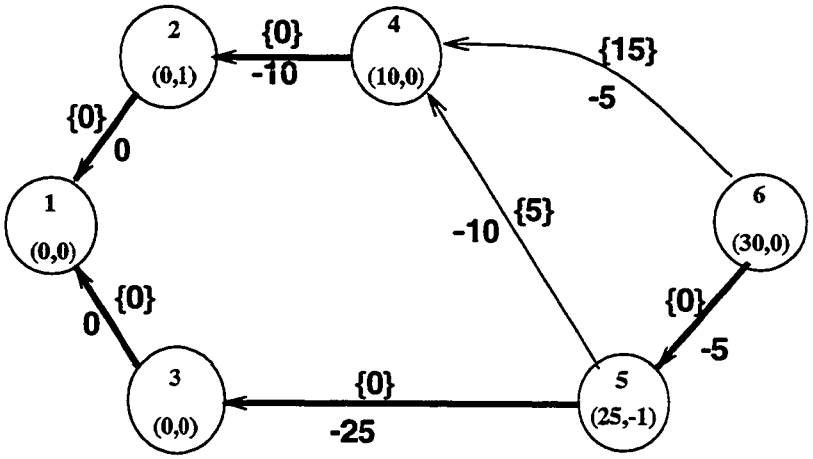

Figure 3.12: Graph G": a bfs where thick edges will form the 0-token spanning tree

nodes. Now, we construct a 0-token spanning tree using a tree-building (ZST-Org) subalgorithm [4].

3.3.3 Building a 0-token spanning tree

Since this phase is entered only upon the successful completion of the Clustering, Contraction and Shortest-path computation subalgorithms at the end of which all nodes in the graph are collapsed into one single cluster, there will be a zero-weighted path between every pair of nodes and a 0-token spanning tree can be built using only zero-edges. Note that there could be more than the required number of zero-edges to build the tree. However, the algorithm chooses only $N-1$ zero-edges required to build the tree.

Each node, v_i, maintains a list of nodes (*ZeroNodes[i]*) adjacent to its outgoing zero-edges. An atomic variable *tree* is initialized to *nil*. Each node in the graph is associated with a flag which is initialized to false. The process to build the tree works in phases. During each phase, all N nodes try to join *tree* by adding their nodes (*ZeroNodes*), however, a node joins the tree only if it or one of its *ZeroNodes* is

already in the *tree*. If there are more than one node already in the *tree*, then any one of them can be considered to avoid forming a cycle in the required 0-token spanning tree. Each node which succeeds in joining *tree* sets its flag to true to indicate that it will be inactive in the next phase.

This subalgorithm terminates when all the nodes in the graph succeed in joining *tree*. The resultant 0-token spanning tree is shown using thick edges in fig. 3.12.

Note that a node which succeed in joining *tree* locks it before updating it, thus making all processes during each phase execute sequentially. Depending on the graph, most of the nodes (processes) during a phase may not find their adjacent nodes in *tree* and end up with doing nothing which makes this approach execute more phases[6] to build a 0-token spanning tree (its effect on the performance of NDS algorithms is discussed in chapter 5).

Next, we proceed to find an optimal solution using the sequential pivoting strategy which works on the edges of the 0-token spanning tree. Here, we seek to improve the *bfs* by performing pivoting step on each eligible edge one by one.

3.3.4 Sequential Pivoting

Given a 0-token spanning tree T, sequential pivoting starts with any edge $e_{ij} \in T$ (either node v_i or v_j should be a leaf node). If this edge satisfies any firing condition, then the firing is performed and the 0-token spanning tree is modified such that the resulting tree has an objective value WX higher (or same, but not less) than before. If no *firing* is applicable to this edge, then both nodes adjacent to this edge are merged to form a cluster. After clustering, if this new cluster is not represented by a leaf node, then another leaf node in the spanning tree is chosen to continue the search for an optimal solution. If this new cluster is represented by a leaf node, then the search process continues from this leaf node. This process continues until all nodes of the tree are merged to form one cluster. At this point, the current basic

[6]N processes per phase.

feasible solution is an optimal solution and the corresponding graph G is optimal. Note that only leaf nodes with positive or negative weight are active during the pivoting step. This process is visualized in fig. 3.13.

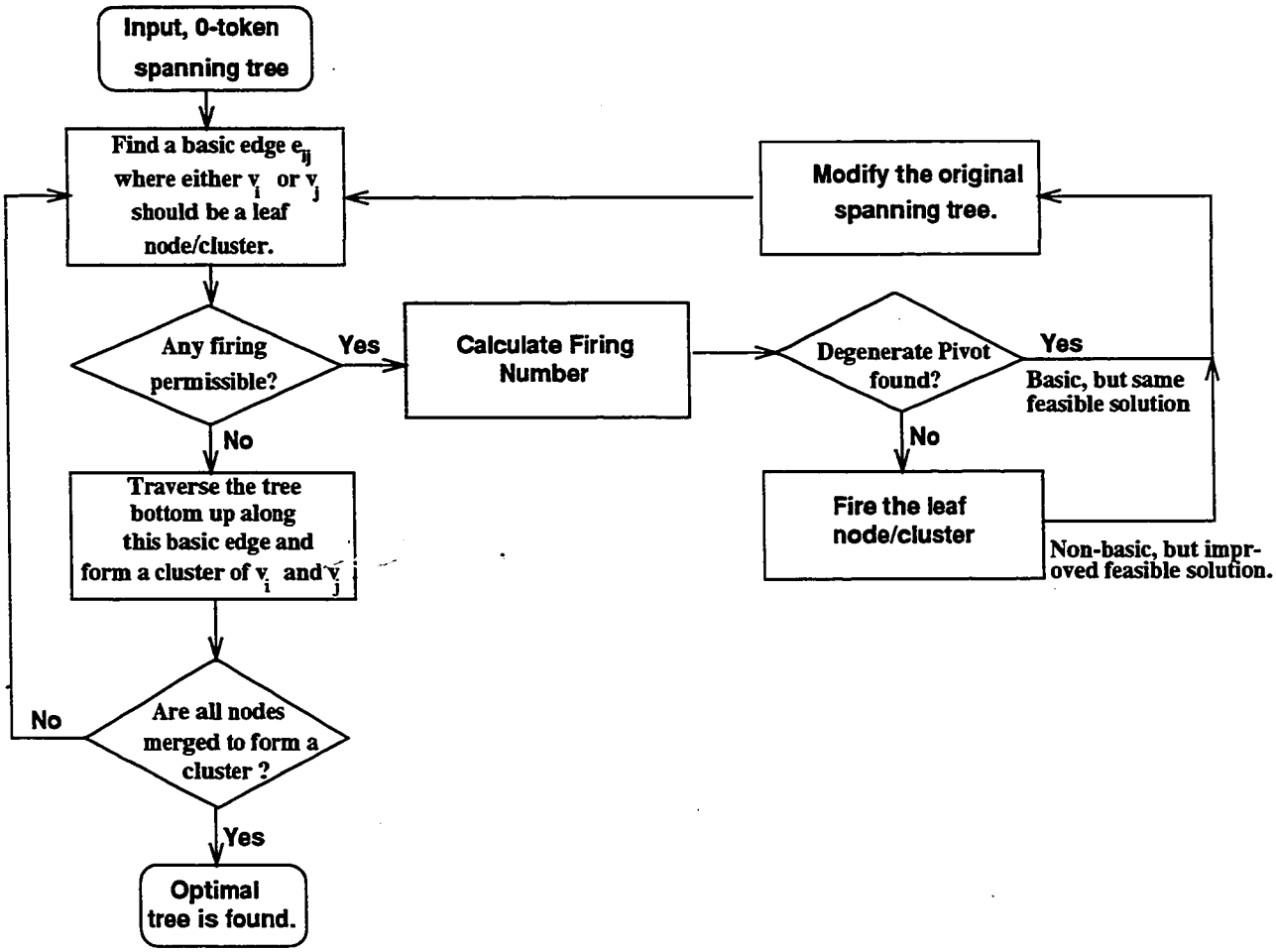

Figure 3.13: Flow of sequential pivoting

To perform pivoting operations, this subalgorithm makes use of the following data structures:

- NumTreeEdges (an array of integers) where NumTreeEdges[i] is the number of nodes that are adjacent to v_i in T.
- Source (an array of integers) where Source[i] is the root of v_i.

- ClusterWeight (an array of integers) where ClusterWeight[i] is the sum of the node weights of all the nodes in the cluster S_i. It is initialized to the weight of each node in the input DTP.

- Tnodes (an array of lists) where Tnodes[i] is the list of nodes that are adjacent to v_i in T.

- FiringNoArray (an array of integers) where FiringNoArray[i] is the number by which element v_i has been fired.

- OriginalInComingNodes (an array of lists) where OriginalInComingNodes[i] is the list of nodes, say k's, for each $e_{ki} \in E$.

- ResidualToken (a 2d-array of integers) where ResidualToken[ij] is the residual token of $e_{ij} \in E$.

Checking for the Firing Condition

Consider a leaf node v_i (or a cluster S_i represented by v_i) of T. If ClusterWeight[i] > 0 and it is connected to the rest of tree by an outgoing edge e_{ij} then it is eligible for positive firing. If ClusterWeight[i] < 0 and it is connected to the rest of tree by an incoming edge e_{ji} then it is eligible for negative firing.

Firing

If v_i is eligible for positive (negative) firing then the firing number is calculated. If it is zero (degenerate pivot) then the tree is modified (see section 3.2) and this operation does not change the solution. If it is not zero, then v_i (node/cluster) is fired appropriately which improves the solution. Note that all nodes in a cluster are fired concurrently. This procedure is described in fig. 3.14. The process of firing a node is visualized in section 3.2 and its procedure (*Fire*) may be found in [4]. The process of *negative firing* is the inverse of *positive firing*. In negative firing, we look

Input:

v_i (Positive firable leaf node/cluster),
Tnodes (0-token spanning tree) and
FiringNoArray,
OriginalIncomingNodes[i],
ResidualToken,
e_{ij} or e_{ji} (basic edge).

Output:

FiringNoArray (Modified firing numbers) and
Tnodes (modified spanning tree).

Procedure Calculate_Positive_firing_number_and_fire.

Begin

{Find a non-basic edge $e_{ki} \in G$ such that it has the minimum residual token. This minimum residual token is used as the cluster firing number f_i and the edge e_{ki} is remembered as the *EnteringEdge*.}

$f_S = \infty$;
For each $k \in$ *OriginalIncomingNodes*[i] **Do**
f_S = Min(f_S, $\text{ResidualToken}_{ki}$)
Endfor
EnteringEdge = e_{ki} with minimum residual token;

{Fire the cluster by f_i and replace the LeavingEdge $e_{ij} \in$ Tnodes by e_{ki} to give modified tree}
ls = all the nodes with root as v_i;
For each $l \in ls$ **Do**
Fire (l, f_S, ResidualToken, FiringNoArray)
Endfor
Tnodes = delete e_{ij} And add e_{ki};

End.

Figure 3.14: Procedure for calculation of positive firing number and firing

look at outgoing edges instead of incoming edges of v_i in the original graph and firing number f_S is negated before we call procedure *Fire*.

Traversing

If v_i is not eligible for any firing then it traverses the tree T along its only adjacent edge e_{ij} (or e_{ji}) to form a new cluster. This is done as described in fig 3.15.

Input:

v_i (a leaf node/cluster of T),
e_{ij} or e_{ji} (basic/zero edge),
Tnodes (0-token spanning tree),
ClusterWeight,
Source and
NumTreeEdges.

Output:

Tnodes (a contracted tree).

Procedure climbUpTree.

```
Begin
        Source[i] = j;
        NumTreeEdges[i] = 0;
        ClusterWeight [j] = ClusterWeight [j] + ClusterWeight[i];
        NumTreeEdges[j] = NumTreeEdges[j] - 1;
        Tnodes[j] = delete v_i from Tnodes[j];
        ls = list of orphan nodes having v_i as their
             representative;
        For each k ∈ ls Do
                ClusterWeight[k] = ClusterWeight[i];
        EndFor
End.
```

Figure 3.15: Procedure for traversing the 0-token spanning tree

The outline of the sequential pivoting subalgorithm is shown in fig. 3.16. Consider the 0-token spanning tree shown in fig. 3.12, the sequential pivoting works as follows. This process constructs a list of all the leaf nodes[7] in the tree and by principle, it can start from any node from this list. Let us start with the leaf node v_4. The weight of this node is zero, so this tree is traversed along its outgoing edge $e_{4,2}$ to form a new cluster as shown in fig. 3.17. To form a new cluster, v_4 performs the following steps:

- source[4] = 2,
- NumTreeEdges[4] = 0,
- ClusterWeight[2] = ClusterWeight[2] + ClusterWeight[4] = 1,
- NumTreeEdges[2] = NumTreeEdges[2] - 1 = 1,
- Remove v_4 from the Tnodes[2],
- ClusterWeight[4] = ClusterWeight[2].

As a result of this process, adjacent nodes of $e_{4,2}$, v_2 and v_4, are merged into one cluster, because the corresponding elements satisfy the minimum spacing requirement relative to each other.

The resulting cluster is represented by a positive weighted leaf node v_2 and it is connected to the rest of spanning tree by an outgoing edge $e_{2,1}$. Thus, this edge satisfies the positive firing condition (see section 3.2). Next, the following steps are performed:

- Both nodes v_2 and v_4 in this cluster look at their incoming non-zero edges, $e_{5,4}$ and $e_{6,4}$, to find the minimum residual token. The non-zero edge $e_{5,4}$ carries the minimum value of five. This value is the firing number of the cluster and this non-zero edge is remembered as the *EnteringEdge*.

[7] If NumTreeEdges[i] = 1 then v_i is a leaf node.

Inputs:

Tnodes (0-token spanning tree),
FiringNoArray,
ClusterWeight,
OriginalIncomingNodes,
OriginalOutcomingNodes,
Source and
NumTreeEdges.

Output:

FiringNoArray (Modified firing numbers).

Algorithm: sPivots

```
Begin
    ls = Collect all the leaf nodes of Tnodes;
    while ls≠∅ do
      {Select an edge e_ij or e_ji ∈ Tnodes}
      leaf_node = v_i i∈ls;
      { outflag indicates that firing is done or not}
      outflag = false.
      while (outflag is false) and (v_i is a leaf node) do
        cw = ClusterWeight[i];
        if (cw>0) and (the adjacent edge is e_ij) then
        begin
          {Positive Firing}
          calculate_positive_firing_number_and_fire (v_i, Tnodes,
          FiringNoArray, OriginalIncomingNodes[i], ResidualToken,
          e_ij) ;
          outflag = true;
        end
        else if (cw<0) and (the adjacent edge is e_ji) then
        begin
          {Negative Firing}
          calculate_negative_firing_number_and_fire ( v_i, Tnodes,
          FiringNoArray, OriginalOutcomingNodes[i], ResidualToken,
          e_ji) ;
          outflag = true.
```

```
            end
            else
               climbUpTree (v_i, e_ij or e_ji, Tnodes, ClusterWeight,
               Source, NumTreeEdges);
            end if
            {Choose the adjacent leaf node.}
            v_i = v_j;
         end while
         {True value of outflag indicates that,
          firing is done and the tree is modified.}
         if (outflag is true) then
         {Re-initialize all the data-structures
          according to modified 0-token spanning tree
          Tnodes}
         end if
         ls = Collect new leaf nodes of Tnodes;
      end while
End.
```

Figure 3.16: Algorithm for sequential pivoting

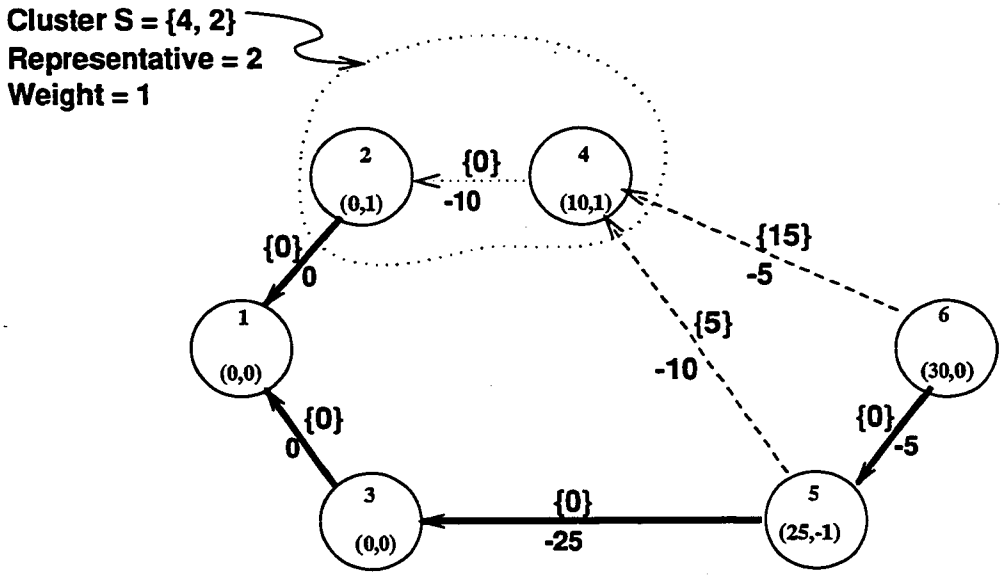

Figure 3.17: First iteration of serial pivoting

- Next, both nodes are fired by five concurrently and residual tokens are updated as shown in fig. 3.18. Each node in that cluster updates its firing number by five. The residual token of outgoing edge $e_{2,1}$ is added by five and the residual token of incoming edge $e_{4,2}$ is subtracted by five. Similar operations are done for node v_4.

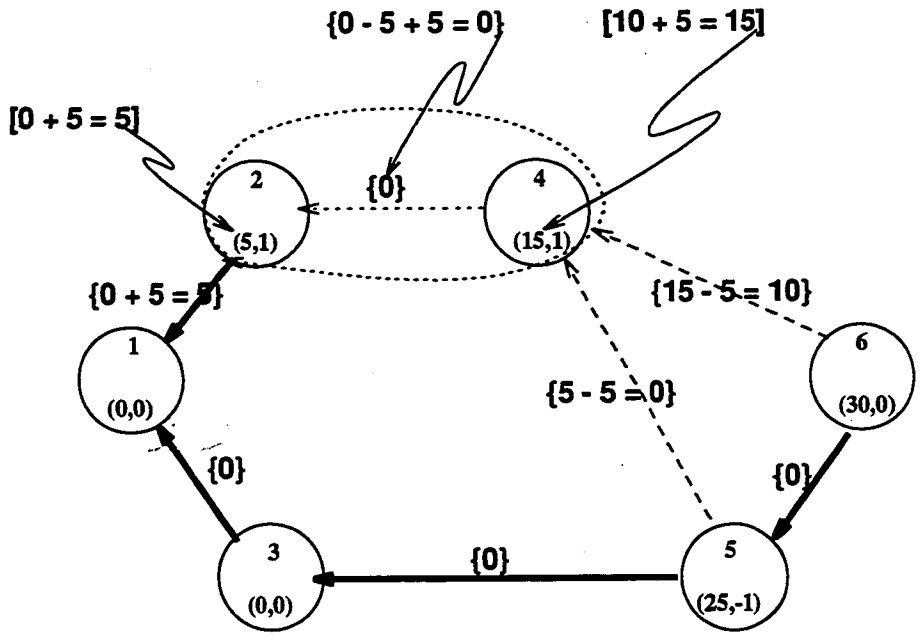

Figure 3.18: Positive firing during second iteration

As a result of this *firing*, corresponding elements are moved by 5 units against the direction of compaction without affecting the width of compacted layout. This also minimizes the overall wire length by four units as shown in the compacted and wire-balanced layout (fig. 3.22). Note that this firing does not cause the residual-token of any edge to become negative. But the tree is no longer basic, because the residual token of $e_{2,1}$ became five as a result of this firing. The next step of this iteration is to maintain the basisity by modifying the tree. To do so, $e_{2,1}$ becomes the *LeavingEdge* and $e_{5,4}$ becomes the *EnteringEdge* as shown in fig. 3.19.

Sequential pivoting algorithm re-starts its operation on modified tree starting from leaf node v_1. Thick dashed arrows, in fig. 3.20, indicate the flow of sequential

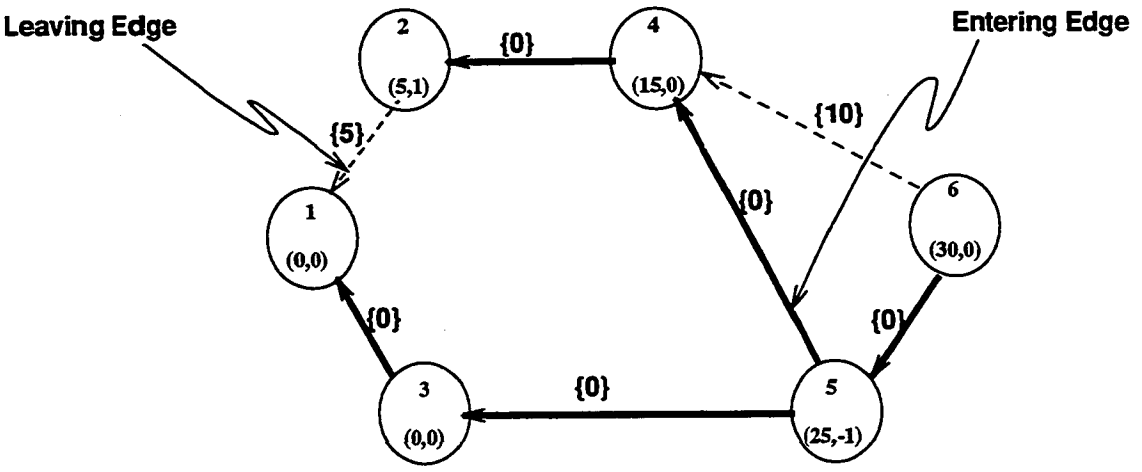

Figure 3.19: Modified tree after positive firing

pivoting algorithm. This algorithm checks the positive firing condition starting from

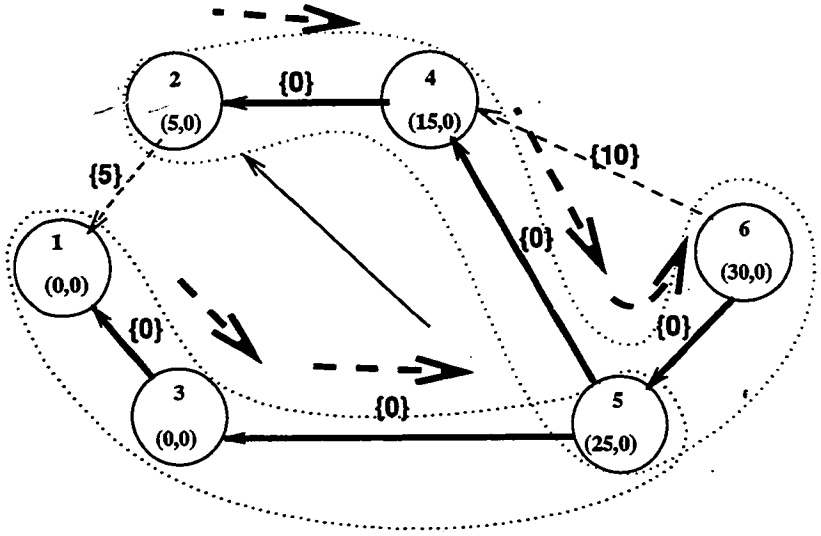

Figure 3.20: Sequential Pivoting on modified 0-token spanning tree

edge $e_{3,1}$, $e_{5,3}$ and then, jumps to $e_{4,2}$ because the cluster of nodes v_1, v_3 and v_5 is not represented by a leaf node (a thin solid line in fig. 3.20 shows an arbitrarily jump from node 5 to node 2). Thus, following this path of thick arrows, serial pivoting merges all nodes into one cluster without any firing and this algorithm terminates with new firing numbers as an optimal solution. The resulting optimal solution X" = {0, 5, 0, 15, 25, 30} with $(WX' = -20) \geq (WX = -25)$. Each $x_i \in X''$ is

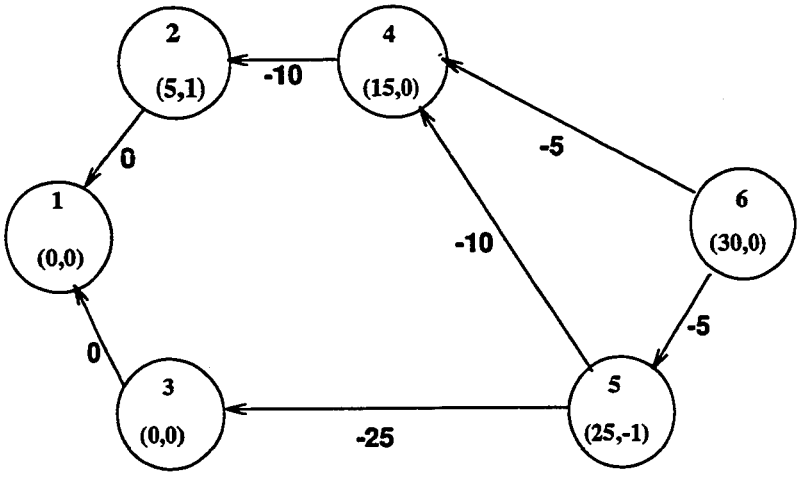

Figure 3.21: Optimal graph at the end of NDS-SP algorithm

the position of corresponding layout feature $L_i \in L$ and it is relative to the left boundary. A total of seven iterations (pivoting steps) are performed to obtain this solution. Note that overall wire length is reduced to ten as shown in fig. 3.22.

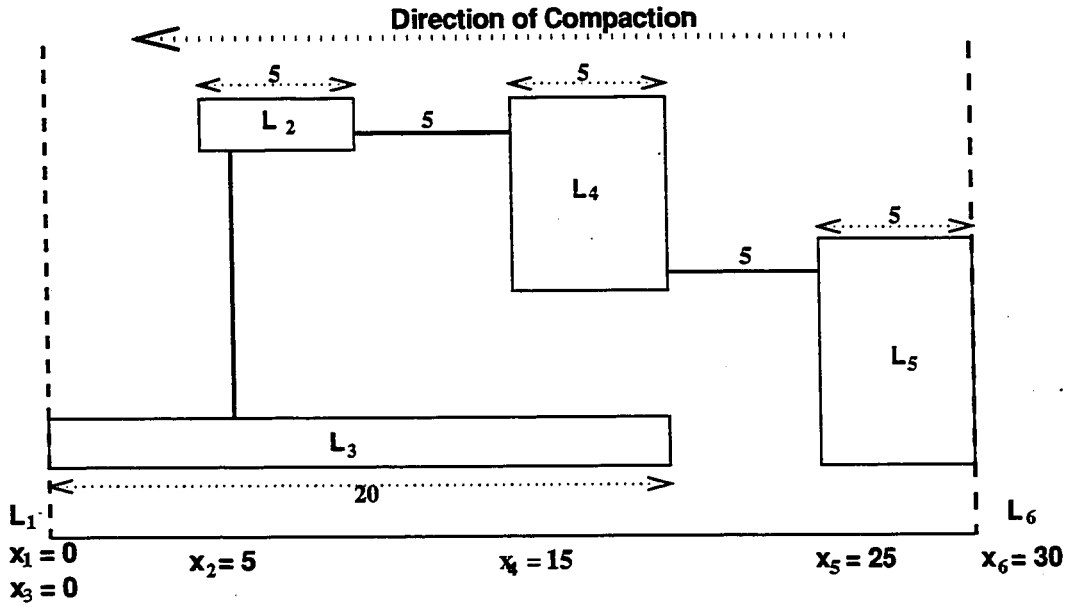

Figure 3.22: Layout after Compaction and Wire-balancing

3.4 The NDS-CP Algorithm

Given a DTP, this algorithm first finds a dual feasible solution, X, by testing the feasibility of this problem as done by NDS-SP algorithm. After finding a dual feasible solution of the DTP, this algorithm constructs a basic feasible tree, T, as described in section 3.3.2. Given T, concurrent pivoting essentially involves traversing the tree bottom up (*starting from the leaves*), identifying the clusters and firing them *concurrently* if residual tokens permit. At the end of concurrent pivoting, the resulting solution may not be *basic*, so this algorithm moves back to find a *bfs* as shown in fig. 3.23. However, while doing so, the objective value WX never decreases [4].

To explain the workings of this algorithm, we use the same graph as shown in fig. 2.4. Starting from a dual feasible solution, all the steps to construct an initial 0-token spanning tree T are the same as explained in section 3.3.2. The next section explains the concurrent pivoting strategy in general and then, the behavior of this strategy on T shown in fig. 3.12.

3.4.1 Concurrent Pivots

After finding T, concurrent pivoting performs simultaneous pivot steps on all the *basic* edges which satisfy the firing condition. After this operation, all leaf nodes traverse the tree along their adjacent zero-edges and merges with the nodes on the other end to form clusters. This results in a smaller tree with new leaf clusters. Therefore, the number of leaf nodes/clusters in each iteration of a concurrent pivots phase decreases. This process goes on until all nodes merge into one cluster. At the end of this phase, two cases arise.

- If any firing is done during an iteration of this phase, then the NDS-CP algorithm moves back to retain the *basisity* as shown in fig. 3.23. The objective value of the resulting solution will be either greater than or equal to previous one because each pivoting operation tries to find a better solution. This

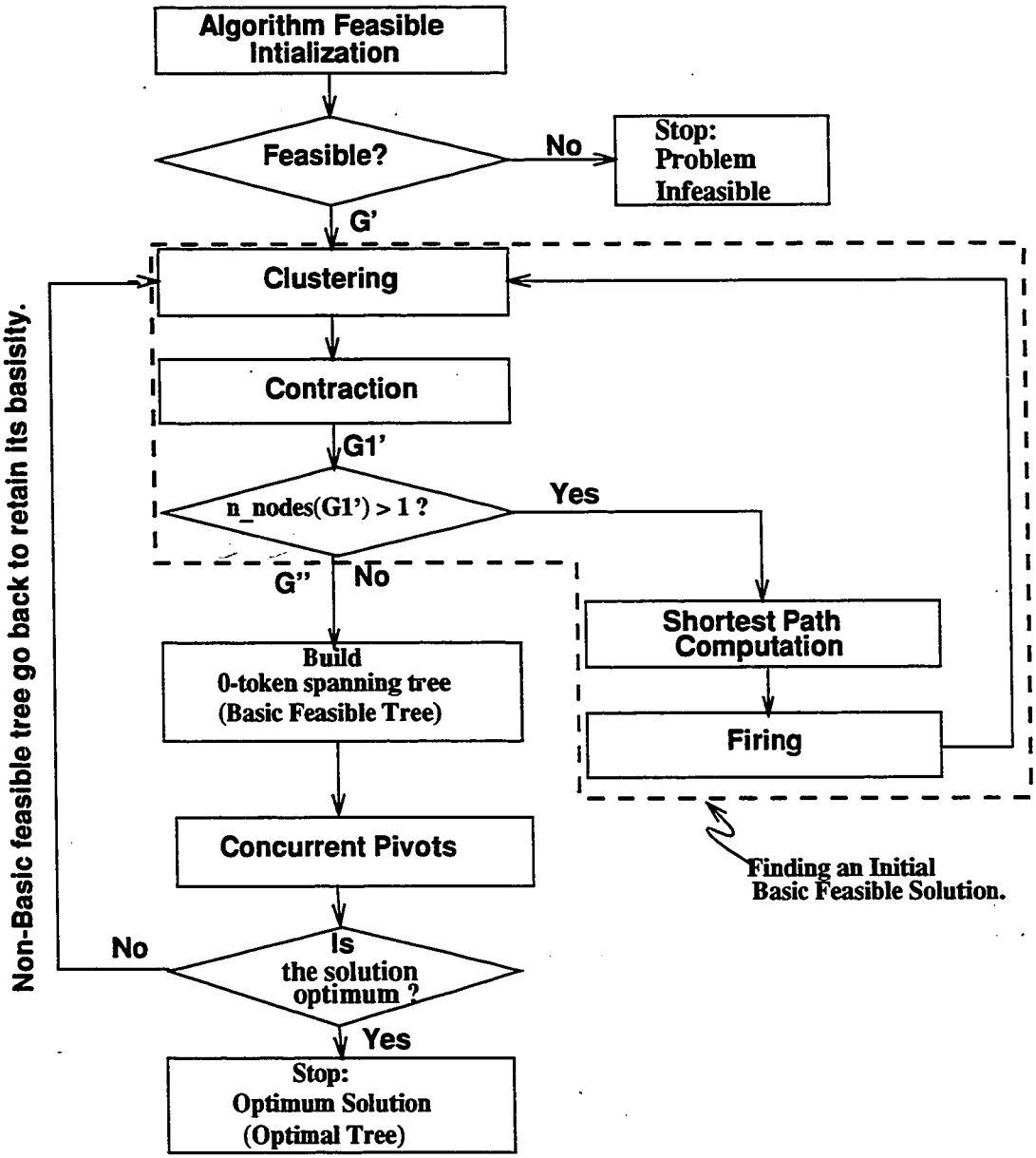

Figure 3.23: Flow diagram for NDS-CP algorithm

process continues until all nodes merge into one cluster without any firing.

- If no firing is done, then the dual feasible tree is an optimal tree.

This flow of concurrent pivoting strategy is visualized in fig. 3.24.

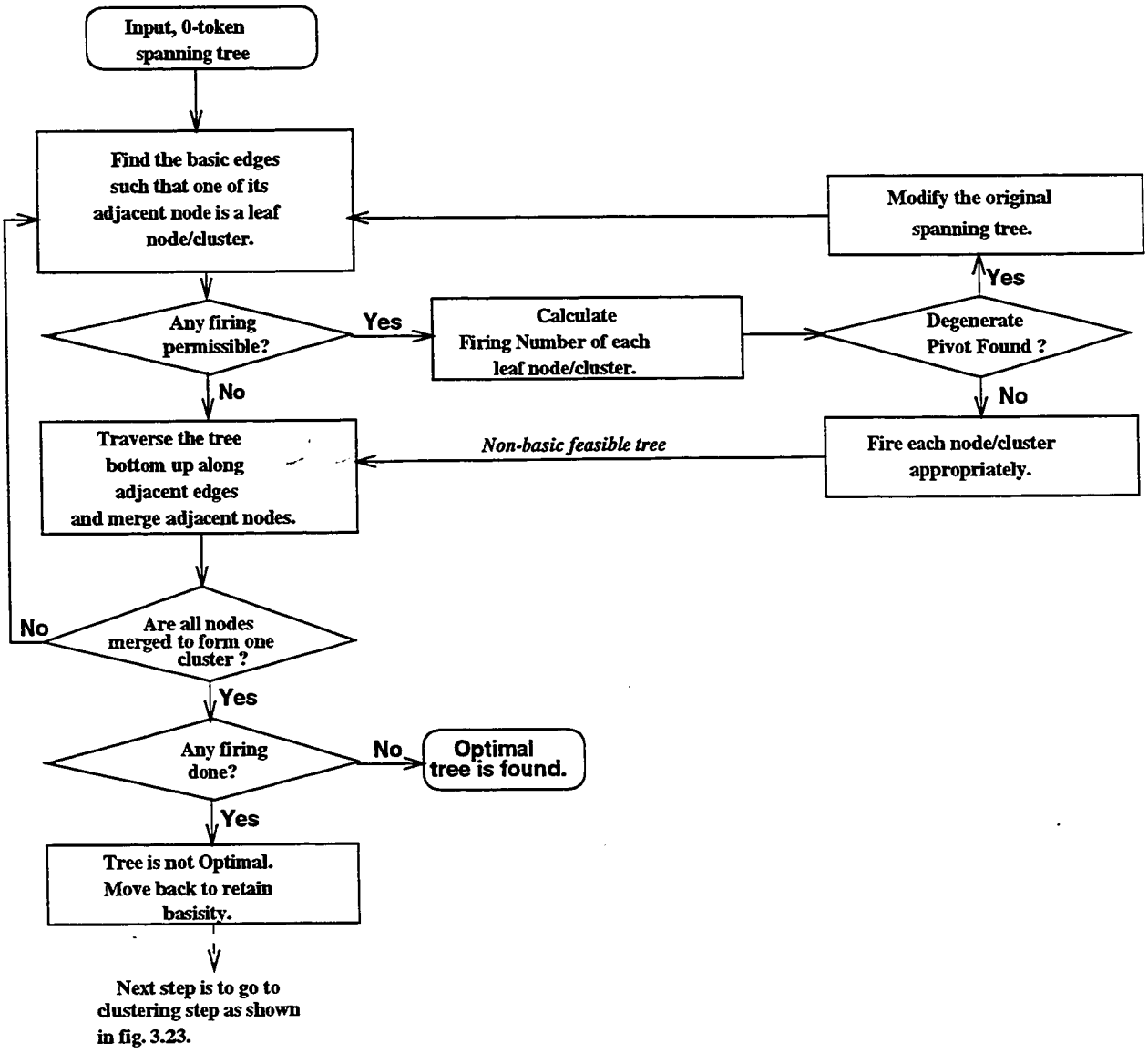

Figure 3.24: Flow of concurrent pivoting

Note that after a firing operation, the concurrent pivoting strategy continues its search for an optimal tree even though the tree has become non-basic. However,

sequential pivoting strategy converts non-basic tree back to a basic tree after each firing operation before it continues its search.

As shown in fig. 3.25, two leaf nodes, v_4 and v_6, are active simultaneously. Their node weights are zero, so the tree is traversed bottom up along their adjacent *basic* edges to form new leaf clusters.

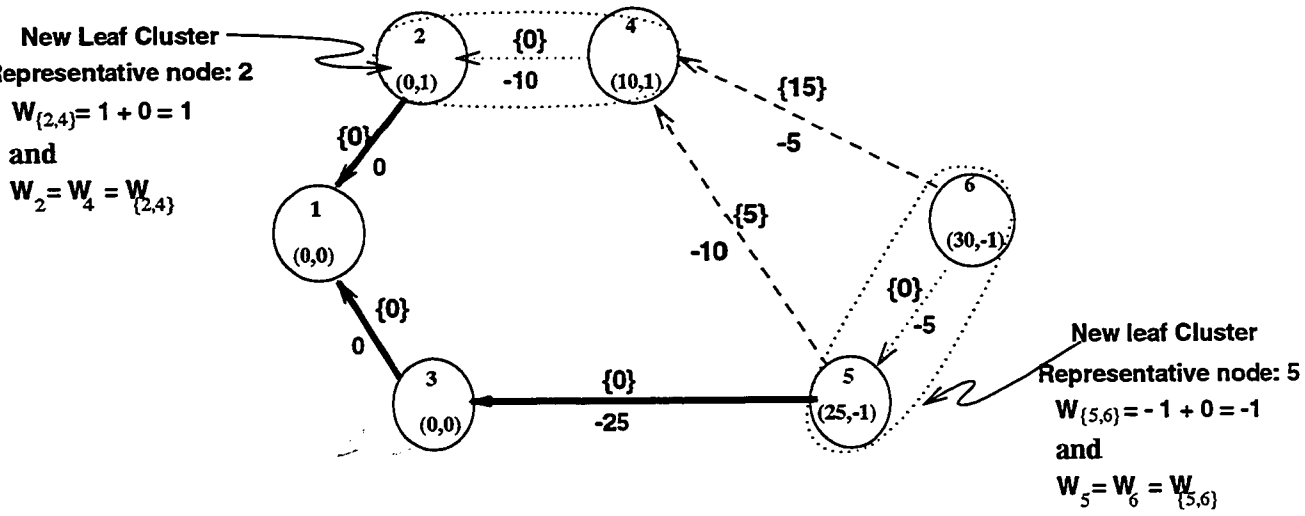

Figure 3.25: First iteration of concurrent pivot strategy

Consider $e_{2,1}$. Removing this zero-edge divides all the nodes into two sets, namely, S = {2,4} and S' = {1,3,5,6}. Note that S is the positively weighted leaf cluster and it is represented by v_2. For $e_{2,1}$ to satisfy the positive firing condition, it should be *leaving* this leaf cluster. Hence, this edge satisfies the positive firing condition. Consider $e_{5,3}$. Removing this zero-edge divides the node set into two sets, namely, S = {5,6} and S' = {1,2,3,4}. Note that S is the negatively weighted leaf cluster and it is represented by v_5. However, for $e_{5,3}$ to satisfy the negative firing condition, it should be *entering* this leaf cluster. Hence, this edge does not satisfy the negative firing condition.

After checking the firing condition during second iteration, all the eligible positive weighted leaf clusters are fired positively and then, all the eligible negative weighted leaf clusters are fired negatively. In this example, we have only one eligible

cluster, S = {2,4}, to fire. The cluster firing number is calculated as five, which is the minimum of the residual tokens on the incoming non-zero edges of all the nodes in this cluster. The firing operation is shown in fig. 3.26. The firing numbers of the

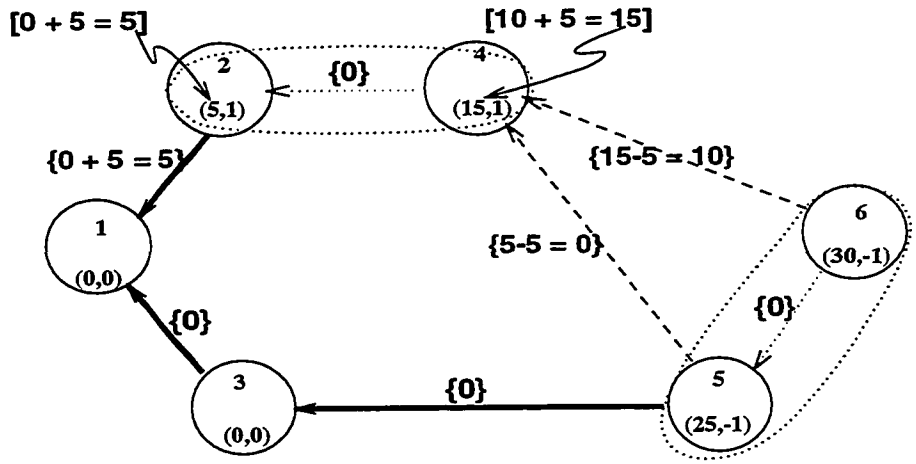

Figure 3.26: Positive firing during second iteration

nodes in this cluster are increased by five and residual tokens are modified. Zero-edge $e_{2,1}$ becomes non-zero and non-zero edge $e_{5,4}$ becomes zero as a result of firing. Thus, the tree loses its *basisity*. However, this tree is **not** modified after firing to retain *basisity*. Note that this operation moves corresponding non-constraining elements, L_2 and L_4, to the right boundary by five units without effecting the compacted width of the layout.

After firing, this non-basic spanning tree is traversed bottom up (starting from leaf clusters) without any firing, and all nodes merge to form one cluster during third iteration as shown in fig. 3.27. This indicates that all elements are at their desired minimum spacing with respect to each other.

As a result of positive firing done during second iteration, this tree loses its *basisity*. So, the algorithm moves back to find a *bfs* and build a new spanning tree. C-C-SPC-F computations on this solution do not improve the objective further. So, the objective function remains the same and the corresponding new 0-token spanning tree is shown with thick edges in fig. 3.28.

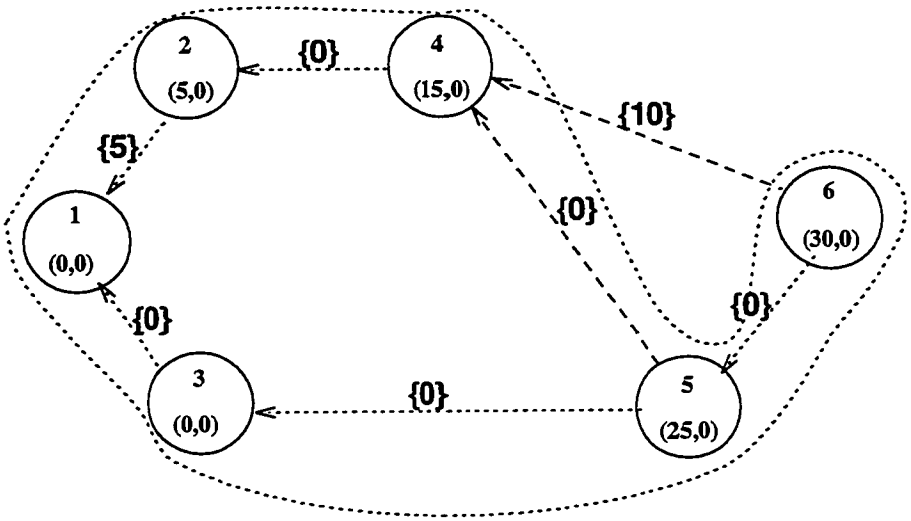

Figure 3.27: Clustering during third iteration

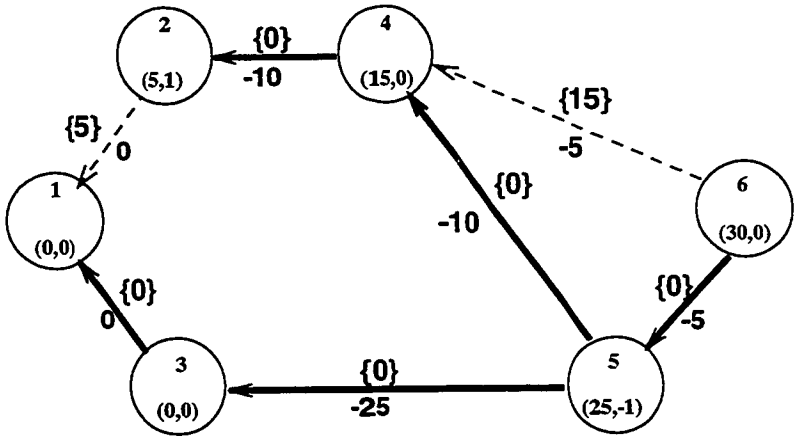

Figure 3.28: Modified spanning tree

Concurrent pivoting strategy is applied again on this tree. During its first iteration, three leaf nodes (v_1,v_2 and v_6) are active simultaneously. However, none

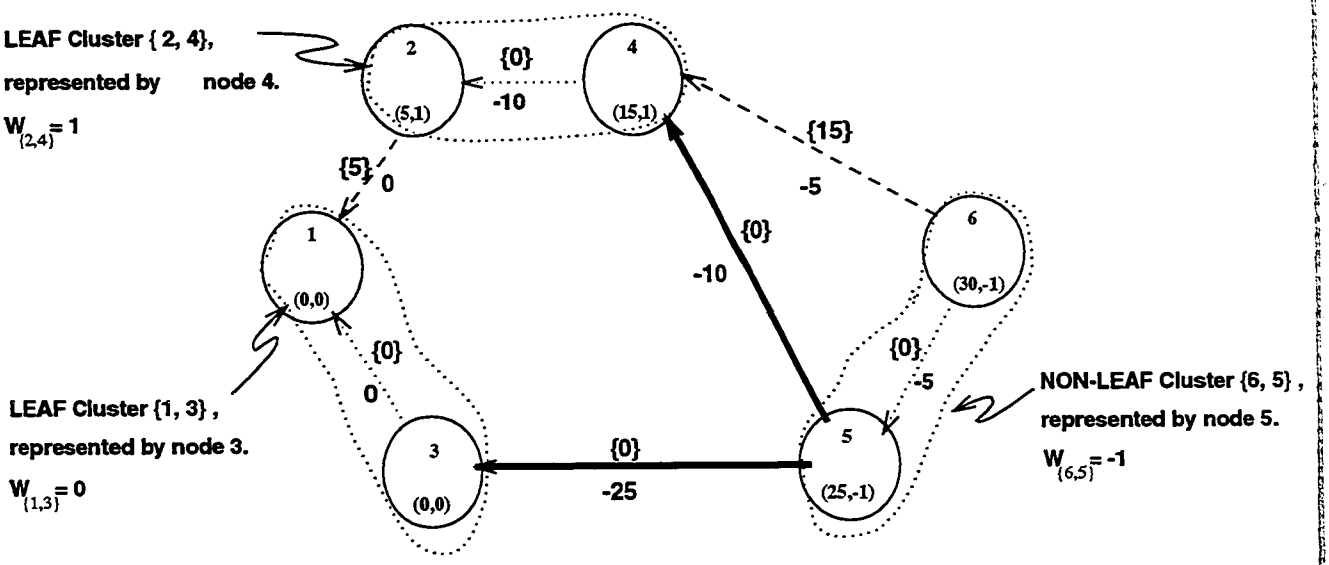

Figure 3.29: Clustering during first iteration

of the adjacent zero-edges is eligible for firing. So, the tree is traversed bottom up to form new leaf clusters as shown in fig. 3.29. This indicates that the elements, with in their respective clusters, satisfy minimum spacing distance.

The second iteration is shown in fig. 3.30. Only new *leaf* clusters became active and the adjacent zero-edges $e_{5,3}$ and $e_{5,4}$ are checked for *firing condition.* However, none of them is eligible for *firing.* Then, this tree is traversed along these edges and a single cluster is formed. Thus, all nodes are merged into one cluster without undergoing any firing operation during this phase of concurrent pivoting. This indicates the optimality of the resulting solution. Here, five iterations (pivot steps) are performed to obtain this solution and we obtain the same solution as obtained with the NDS-SP algorithm. The corresponding optimal solution and graph is the same as shown in fig. 3.21.

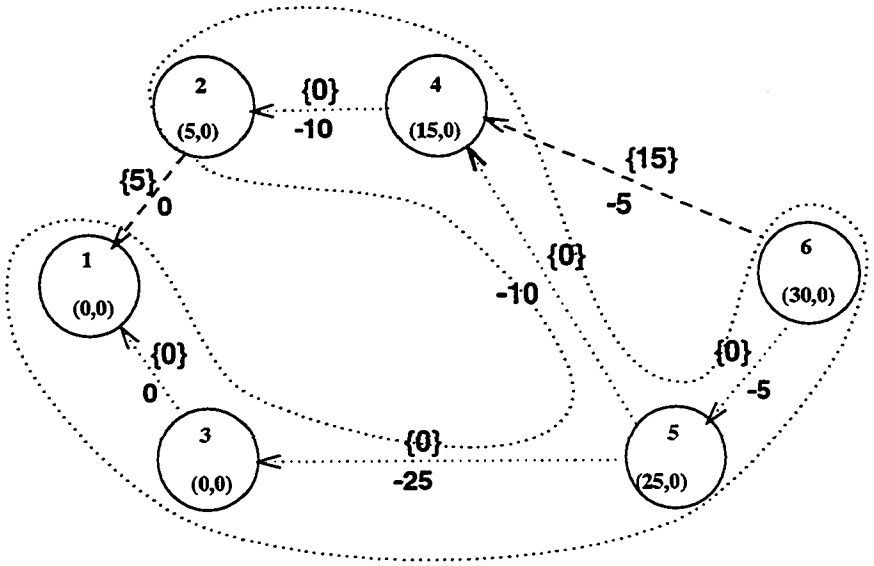

Figure 3.30: Clustering during second iteration

3.5 Integrated LCWB

During an optimization step in the solution of the LCWB problem, the constraining elements may be moved and their movement may increase the compacted width of the layout even though the overall wire length is minimized. Their movement can be restricted after a compaction step to maintain the compacted width. This results in a minimum layout width but the overall wire length may not be minimum [4].

Note that after the application of algorithm Feasible, the residual tokens of the edges on the longest path will be zero (see fig. 1.3 and fig. 3.7). Let S_l denote the set of these critical nodes, x_i is the position of the corresponding element L_i after the application of this subalgorithm and d_{lr} is the length of this path. To keep each node $v_i \in S_l$ at x_i and then, adjust the x_j of the nodes $v_j \notin S_l$, we add an artificial edge, e_{rl}, to the constraint graph G before we start the optimization step. This edge is assigned a weight of d_{lr}. In this way, the overall wire length is minimized without effecting the width of the compacted layout.

3.6 Zero-Token Spanning Tree

In the original approach [4], a zero-token spanning tree is built using depth-first search of the subgraph of zero-edges and is discussed in section 3.3.3. This approach is insufficient due to the fact that one node joins the tree at a time which makes it sequential in nature. Thus, it becomes the critical step in the NDS-CP algorithm because most of the time is spent in building the tree than doing other operations (see chapter 5). In order to improve the performance of the NDS-CP algorithm and NDS algorithms in general, we developed two new methods to generate the tree. In this section, we present these subalgorithms.

3.6.1 Algorithm: ZST-SM

In this approach, called 0-token Spanning Tree using Sequential Merging (ZST-SM), there are two passes.

- During pass one, each node builds a subtree of its outgoing zero-edges. By doing so, it builds a complete 0-token spanning tree but this tree may have cycles[8] in it.
- During pass two, we allow each subtree to delete all but one zero-edge by which it is connected to the rest of the 0-token spanning tree to remove such cycles.

Pass two proceeds in phases. During any phase of pass two, a node may interact with others while accessing a global *tree* or its neighbour's *Tnodes* structure. The global *tree* structure makes a subtree join it one at a time. However, this approach performs better than the original approach due to the parallelism in pass one. The outline of this subalgorithm is given in fig. 3.31.

[8]If a subtree is connected to another subtree of the 0-token spanning tree with more than one zero-edge, then they form a cycle.

Input:

ResidualToken (2-d matrix of residual tokens)

Output:

Tnodes (0-token spanning tree; Tnodes[i] is the list of incoming and outgoing nodes to and from a v_i.)

Algorithm: ZST-SM

```
begin
  {Initialize Tnodes[i], SubTrees[i] to nil ∀ i∈N.}
  {Initialize JobDone[i] to flase ∀ i∈N.}

  {Pass one: Build Subtrees.}
  for each v_i ∈ G do
    {Following list is constructed using ResidualToken}
    ZeroNodes = A list of nodes adjacent to v_i's outgoing
                zero-edges.
    if ZeroNodes≠∅ then
      JobDone[i] = false;
    else
      JobDone[i] = true;
    endif
    SubTrees[i] = ZeroNodes∪i;
    Tnodes[i] = Tnodes[i]∪ZeroNodes;
    for each v_j ∈ ZeroNodes do
        Tnodes[j] = Tnodes[j]∪i;
    endfor
  endfor

  {Pass two: Sequential Merging.}
  NotFinished = true;
  ROOT = v_i ∈ G;
  while NotFinished do  {start a phase.}
    tree = ∅;
    {Construct a list of subtrees.}
    ls = list of nodes v_i such that JobDone[i]≠true;
```

```
    if ls = ∅ then
        NotFinished = false;
    else
        NotFinished = true;
    endif
    for each v_i ∈ ls do {It is a Doacross loop.}
        if v_i is ROOT then
            tree = SubTrees[i];
            JobDone[i] = true;
        else if (SubTrees[i] ∩ tree) ≠ ∅ then
            tree = tree ∪ SubTrees[i] ;

            {Collect all adjacent nodes by which subtree
             v_i is connected to others already joined tree.}
            ListOfConnectingEdges = (SubTrees[i] ∩ tree) ;

            {Remove the first node from ListOfConnectingEdges.}
            EdgesToBeDeleted = getTail ListOfConnectingEdges;

            for each v_j ∈ EdgesToBeDeleted do
              Delete v_j from Tnodes[i];
              Delete v_i from Tnodes[j];
            endfor
            JobDone[i] = true;
        endif
    endfor
  endwhile
end.
```

Figure 3.31: Zero-Token Spanning Tree using Sequential Merging

3.6.2 Algorithm: ZST-HM

The first pass in this approach is to construct the subtrees as done in ZST-SM. The second pass in this approach method merges the subtrees in a hierarchical fashion.

In the second pass, pairs of subtrees are formed, and we assign each pair to a process. During each process, if both subtrees in a pair are merged to form a new subtree, then the root of one of the subtree becomes the root of new sub-tree and the other root becomes the child. As a result, the number of subtrees reduces after each phase of the second pass. If a pair does not merge, then both subtrees in that pair will be active during the next phase.

Next, we collect the new roots (subtrees) for the next phase. If the number of subtrees is not less than the previous phase, then we permute the subtrees in order to form new pairs. This goes on until all subtrees collected in the first pass are merged to form a required 0-token spanning tree.

All phases are executed sequentially, however, all pairs during a phase try to merge concurrently. However, the pairs which do not merge effect its performance. The performance results (discussed in chapter 5) show that ZST-HM method behaves much better than the original approach to build the tree. The outline of this subalgorithm is given in fig. 3.32.

3.7 Summary

In this chapter, we explained the working procedure of the NDS-SP and NDS-CP algorithms on a small graph. For the 6-node DTP in our example, the NDS-SP algorithm executes one iteration of the optimization phase and performs seven pivot operations to find an optimal solution, however, the NDS-CP algorithm executes two iteration of the optimization phase and performs three pivot operations during its first and two pivot operations during its second iteration to find the same solution. For the 50-node graph we used in our experiments, the NDS-SP executes

Hierarchical Merging.

```
begin
   NotFinished = true;
   {Initially each node is the source of itself.}
   SubLists[i] = nil ∀ i ∈ N/2;
   Source[i] = i ∀ i ∈ N;
   {A list of the subtrees}
   ls = list of v_i such that JobDone[i] ≠ true;
   while NotFinished do {start a phase}
      { Construct pairs of subtrees from ls;
        SubLists[i] will consist of a list of two subtrees;}
      ListOfIds = a list of i such that SubLists[i] ≠ ∅ ∀ i ∈ ls;
      for each i ∈ ListOfIds do
       SubLs = SubLists[i];
       Tree = ∅;
       ROOT = v_i ∈ SubLs;
       for each v_j ∈ SubLs do
        if v_j = ROOT then
           Tree = SubTrees[j];
        else if (SubTrees[j] ∩ Tree) ≠ ∅ then
              Tree = SubTrees[j] ∪ Tree;
              ListOfConnectingEdges = SubTrees[j] ∩ Tree;
              EdgesToBeDeleted = getTail (ListOfConnectingEdges);
              {Collect all the nodes represented by v_j.}
              tmpls = List of v_k such that Source[k] = j ∀ k ∈ N;
              for each v_x ∈ EdgesToBeDeleted do
               for each v_y ∈ tmpls do
                if v_x ∈ Tnodes[y] then
                  Delete v_x from Tnodes[y];
                  Delete v_y from Tnodes[x];
                endif
               endfor
              endfor
              {The second subtree becomes the child.}
              Source[j] = ROOT;
              {A new subtree is formed.}
              SubTrees[ROOT] = Tree;
        endif
       endfor
      endfor
```

```
    {Each node in G corrects its source information.}
    for i = 1 to N do
      if Source[i] = i or Source[i] = 0 then
        Source[i] = Source[Source[i]];
      endif
    endfor
   {Construct a new list of SubTrees.}
    ls = list of nodes such that Source[j] = j ∀ j∈ls;
    if ls is ∅ then
      NotFinished = False;
    else
      { count the number of elements of ls, say cnt. If cnt
        is not less then the previous phase of merging,
        then permute this list. }
    endif
  endwhile
end.
```

Figure 3.32: Hierarchical Merging, the second step of ZST-HM

one iteration of the optimization phase and performs 476 pivot operations, however, the NDS-CP algorithm executes three iterations of the optimization phase and performs 22 pivot operations during its first, 8 pivot operations during its second and 1 pivot operation during its third iteration. We observe that the concurrent pivoting strategy, an approach to parallelize the sequential pivoting strategy, is successful in decreasing the number of pivot operations. However, the loss of basisity of the solution after performing concurrent pivot operations, and then retaining it using the repeated processes of constructing a dual feasible tree and concurrent pivot operations makes it slower than the NDS-SP algorithm. This will be described in detail in chapter 5. We also discussed the solution of ILCWB problem using these algorithms. Lastly, we described two new methods to build the 0-token spanning tree which are investigated to enhance the performance of these algorithms.

Chapter 4

Overview of Id and Monsoon

Id is a high-level programming language founded on the principles of modern FP and *dataflow execution*. In order to describe its features, which allows us to write parallel programs without worrying about the details of parallelism, we need to understand its execution model which is based on dataflow execution.

In section 4.1, we describe the execution model for *Id* programs. In section 4.1.1, we briefly describe the Monsoon hardware. Then, the language features which help in expressing the parallelism in our *Id* programs are described in section 4.3.

4.1 Parallel Execution Model

In sequential programming languages, textual code order defines execution order. A line of code that textually precedes another will execute first. In implicitly parallel languages, however, code order does not define execution order. Execution order is constrained only by data dependencies which occur when the result of one instruction is an argument to another instruction. When an instruction's data is available, that instruction is ready to execute. Consider the following example describing a data dependency written in Id.

```
def test n =
{ b = a * 2;
  a = n + n;
 in
   b};
```

In this example, the first line can't execute until the second line has completed execution. The *multiply* on the first line waits for the *add* on the second line to complete since it needs the value a produced by the second line.

Consider an implementation of *fibonacci* in *Id.*

```
def fib n =
  if n < 2 then 1
  else
    fib (n-1) + fib (n-2) ;
```

Because there are no data dependencies between the recursive calls to *fibonacci*, it is possible for both recursive calls to be active simultaneously. Id's *non-strict semantics* allow many procedures to be active simultaneously. Any procedure may have one or more active child procedures because it permits the execution of a child procedure to begin even before all of its arguments have been computed. Child procedures do not have to terminate in the same order in which they were invoked.

The basic parallel execution model is shown in fig. 4.1. All of the activation frames[1] (not just the leaves) in the call tree are potentially active. When a procedure finishes, it is guaranteed to be a leaf in the activation tree because all of its children must already have completed, and thus, its frame is removed from the activation tree and reclaimed for reuse.

[1]It is a block of contiguous memory locations. Every invocation of a procedure is associated with such a frame.

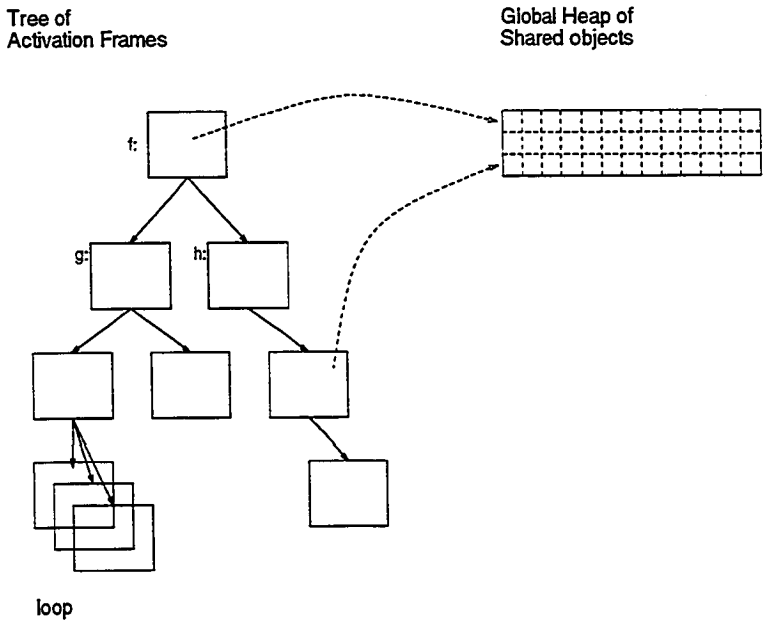

Figure 4.1: Fully parallel execution model

4.1.1 Dataflow Execution

Consider an expression $v = (a * x) - (b + y)$. Its dataflow graph, shown in fig 4.2, consists of three operators (actors). Each operator represents an instruction and each arc indicates the flow of a value. With the exception of external inputs and the output arcs, an arc originates at the instruction that produces the value and ends at another instruction that uses this value as input operand. The first token to arrive at a binary instruction waits for the other input to arrive. When all the input operands of a particular instruction have arrived, the corresponding instruction is enabled for execution. This *dataflow style* of execution is also called *data-driven* execution.

Monsoon employs the data-driven program execution model. Since the execution is driven only by the availability of operands at the inputs to the actors, there is no need for a *program counter* in this architecture, and its *parallelism* is limited only

by the data dependencies in the program. The machine works by continuously processing tokens[2] which deliver data to the instructions. Monsoon uses *frame memory* which is local to each processing element (PE), to support data-driven execution. Each binary instruction to be executed is assigned a slot of frame memory.

As shown in fig. 4.2, each node has a field that contains its compiler ordained frame slot offset[3]. A part of frame allocated to run this expression at some instance

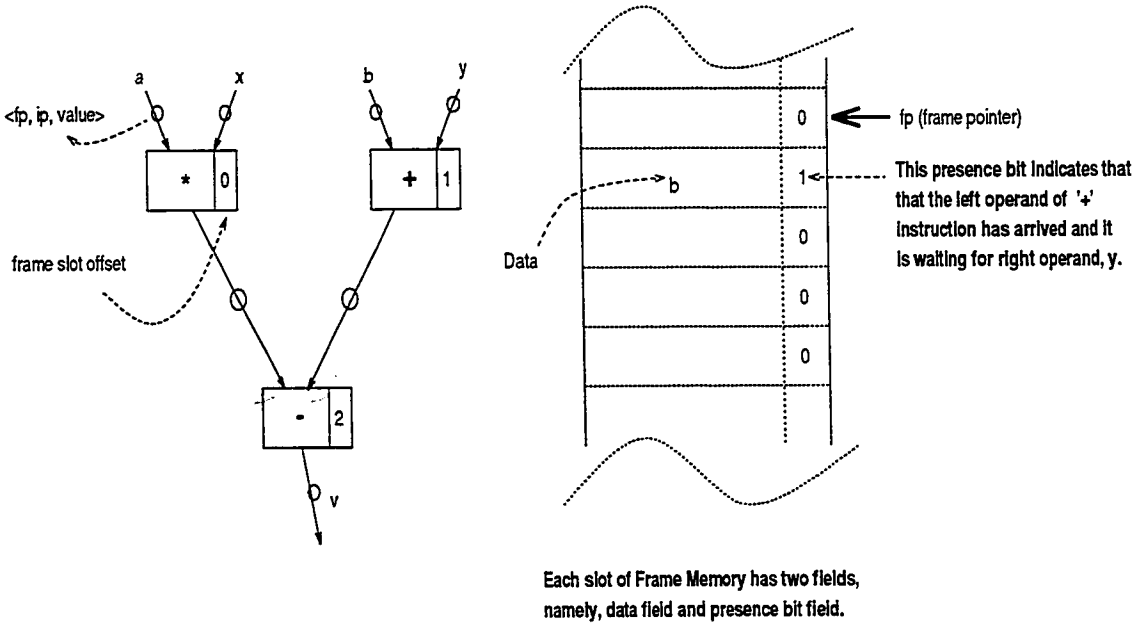

Figure 4.2: Dataflow representation of an expression: v = (a*x) - (b+y)

during execution is also shown in fig. 4.2. Of the two operands of + instruction, **b** has arrived and is waiting for **y**. It can be inferred by looking at frame slot *(fp + 1)*, which is allocated to the + instruction. In frame memory slot, the presence bit is set to indicate that the left operand has arrived, while the value of **b** is stored in the data field of the slot. When **y** arrives, the slot is checked for the presence of **b**, since it is there, the + instruction may execute.

[2]Tokens are computation descriptors. These are comprised of two parts: *the continuation* and the *data*. The former consists of an instruction pointer, a processor number, left or right operand specification, and a frame pointer (fp).

[3]The offset of a binary instruction's matching frame slot relative to **fp** is determined at compile time.

The frame pointer, *fp*, distinguishes between values destined for the same instruction and operand port that come from different invocations of the same (possibly recursive) procedure; that is, each invocation of the procedure is allocated a different frame.

This dataflow-style of execution allows Monsoon to exploit the parallelism inherent in an Id program.

4.2 Monsoon: A Dataflow Processor

Monsoon [PC90] is a shared memory multiprocessor consisting of a mixture of processors (PE) and I-structure (IS) units, interconnected by a packet-switched network. Each PE is an experimental eight-stage pipelined parallel processing node. It runs at 10 MHz, and is capable of processing up to ten million tokens per second.

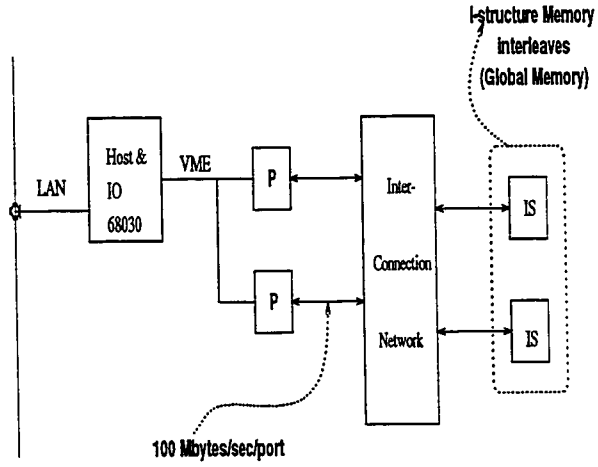

Figure 4.3: A shared memory Monsoon dataflow multiprocessor

Monsoon supports *I-structure* units, the memory for large arrays and other data structures, to synchronize producer and consumer of a location in its *hardware*. Global Memory is implemented with I-Structure units. Access to global memory always occurs over the network, and every global read or write (remote request) is structured as a split-phased transaction so that multiple requests may be in progress

at one time, i.e., an instruction issues a request to the processor or global memory module containing the desired data, and then executes other instructions which do not depend upon the result of the request in progress. An I-structure has the property that when a read-request arrives before the write to that location, it is deferred until that element has been written. Once the value is present, it can be sent to the requesters. This synchronization allows us to write deterministic programs, and the split-phased nature of remote requests is used to hide the network latency. The I-structure operations can be simulated in the Monsoon processor's frame memory because it also has presence bits. This allows the compiler and the run-time system to allocate some heap objects locally. Reading and writing to I-structures located in processor memory requires the second phase of the split-phased transaction to be executed on the processor containing the structure. Normally the second-phase operation executes on an I-structure unit.

Monsoon also adopts M-structures as a memory model. M-structures are the set of memory operations for updating shared data atomically. M-structures operation are not embedded in a data abstraction construct and allows a programmer to construct abstractions appropriate to its application, with the ability to control synchronization and scheduling.

Idles

A Monsoon processor has two local token queues (user and system buffers) to hold the tokens. The execution of an instruction results in the production of zero, one or two tokens at the end of eight cycles. These newly created tokens are either circulated to the head of the processor pipeline to be executed next, sent to the network to be delivered to the destination processor's queue or placed in the local token queues. A single processor Monsoon may execute an idle cycle:

- If no recirculating token was produced at the end of the pipeline, it pops a token from its own token queues to execute. If no token is available, then the

processor *idles* until a token is available.

- If an instruction produces only a network token[4] , then there will be an idle cycle because of a structural hazard [5]. In the worst case, the number of idles due to the hardware hazard are estimated to be roughly 1.5 times the sum of the number of fetches and the number of remote send operations.

A Monsoon processor's token queues may be empty due to lack of work. A lack of work occurs due to the following reasons.

- lack of parallelism due to RTS sequentialization,
- lack of parallelism during startup and termination of program, or
- lack of parallelism in the algorithm.

The run-time system can artificially reduce parallelism by taking a long time to satisfy a frame or heap object request. A typical RTS request takes 31 or 32 instructions which are executed sequentially. On Monsoon, the actual latency will be eight times as long because of processor pipeline interleaving.

During the startup and termination phases, there is not enough parallelism to keep a processor busy. The length of the startup and termination phases are affected by the rate at which the program exposes parallelism and the latency of RTS requests. This cost is non-zero on a single processor due to the 8-way interleaved pipeline.

Finally, a lack of parallelism could be caused by the compiler, poorly chosen loop bounds, the size of problem to run, or the algorithm itself (because the subalgorithms used in a program may be inherently sequential in nature).

[4]Most reads destined for the I-structure units generated by the Id compiler create a network token, but no local token, forcing the structural hazard to insert an idle cycle in the processor pipeline.

Bubbles

Monsoon accepts only one token at a time for processing. The first token of a binary instruction always causes the execution unit of the processor to idle. This idling of the execution stages is called a *bubble* in the pipeline.

4.3 Id Language Overview

Id is a layered language with a purely functional kernel, a deterministic layer with I-structures, and a non-deterministic layer with M-structures which are updatable data structures with fine-grain synchronization. Some of its attributes are as follows:

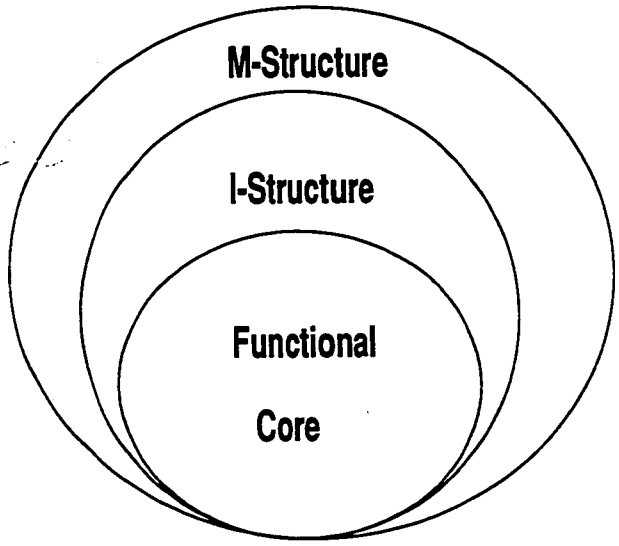

Figure 4.4: Three layers of Id

- Id is a functional language with *implicit parallelism*, i.e., Id programs specify a *partial order* on operations, constrained only by data dependencies. The compiler exploits this property to automatically generate parallel programs.
- Id programs are declarative, meaning that computations are specified as expressions rather than sequence of instructions. Id programs can be written as a set of definitions without any explicit control flow.

- Id has non-strict semantics, meaning that a data structure may be used before all its components are written, and a function may compute and return a result even when some arguments have not been evaluated. *Non-strictness* is used to increase parallelism through *eager evaluation*, in which functions begin computing as soon as they are called, in parallel with the evaluation of their arguments.

- Id is augmented with data structures (M-structures) that can be atomically updated and their atomicity is implicit.

This section is concerned with the language's functional/non-functional features. We also discuss the parallelism contributable by each feature. These features are, namely, single-order and high-order functions, list and array comprehension, conditional statements, recursive procedures, loops, blocks and Mutable data-structures. A detailed description of Id may be found in [10].

4.4 Functional layer

The core of Id is a non-strict functional language with implicit parallelism. This section deals with the features of the *functional layer* used in our implementation and points out the *expressive power* and *parallelism* due to them.

4.4.1 Functional Abstractions

In Id, the computational abstractions are expressed using functions. A function takes some data objects and produces new data objects from them. The method used to produce the new objects from the arguments is abstracted by the function. High-order functions generalize this further by taking data objects as well as other functions as arguments and producing new data objects and new functions as results.

This language directly supports array and matrix structures and provides efficient parallel iterative constructs for their manipulation.

Single-order and High-order functions

In Id, we can define a single-order function as follows:

```
def foo? b =
  (b < MAXINT) or false ;
```

We have defined a *boolean* function, *foo?*. If the integer value b is less than MAXINT value, then this function returns *true*, otherwise *false*.

A function may have any number of arguments. It acts on these arguments without modifying them and produces a result. Thus, there are no side-effects after applying the function to its arguments.

A function is invoked in parallel with the evaluation of its arguments (evaluation of arguments initiates whether they are needed or not). Thus, the computations in the function body can be overlapped with the computation of the arguments. Moreover, a function may return a result before any of its arguments are evaluated.

In Id, a function can also be passed as an argument to the other function. Consider a high-order function *getONodes*. This function constructs a *functional array* using *array comprehension* constructor. In this function, each node i finds all its *outgoing* nodes by calling *outNodes*, and *incoming* nodes by calling *inNodes* function. This function can be read as the array of lists where i increments from 1 to n. $i \leftarrow 1$ *to* n is called the generator and it is bounded to 10 to limit the degree of unfolding to a fixed number of iterations. Each list is constructed by calling the function which is received as an argument. After constructing an array, it can be assigned to *OrigOutNodes* or *OrigInNodes* as shown below.

```
def getONodes n token func =
  {array (1,n) of
```

```
    | [i] = func i n token || i <- 1 to n bound 10
  };

OrigOutNodes = getONodes n token outNodes ;
OrigInNodes = getONodes n token inNodes ;
```

Non-strict semantics of array comprehension allows it to return the array as soon as it has been allocated, with consumers (those expressions using *OrigOutNodes* or *OrigInNodes* in the program) automatically waiting for the individual elements to be assigned. Thus, *producer-consumer* parallelism can be expressed with array comprehension. Moreover, the functions (single and high-order) are major features which give the expressive power to Id.

Blocks

Consider a *block* in *Id*,

```
y1 = { x1 = e1 ;          % Binding
       :
       xN = eN
     In                   % Keyword
       eBody } ;          % Body which is an expression
```

All *bindings* $e1 \ldots eN$ and *eBody* are evaluated in parallel and the value of *eBody* is returned as soon as it is available (even though $e1, \ldots, eN$ may not have finished yet). There is no implicit ordering on the evaluation of bindings, except as imposed by data dependencies. The *block* as a whole is an expression, i.e., it represents a value of the *body*. The expression, *eBody*, can be considered as a *query expression*, i.e., an expression that uses these bindings to produce the desired answer.

In a program where function *f* is defined as :

def f x1 x2 x3 ... xN = eBody ;

The Id compiler may reduce the application (y1 = f e1 e2 e3 e4 ... eN) to the parallel block.

Conditional Expressions

Consider the following function.

```
def foo? b =
  if (b < MAXINT) then true else false ;
```

The interesting feature of *Id* is to avoid evaluation of any arm of the *if-then-else* expression until the predicate is evaluated. This evaluation is also known as an implicitly parallel lenient evaluation order [14]. This is done to increase the efficiency of Id programs.

Parallel Loops

Consider the following function.

```
Def union l1 l2 =
  { set = l2
  in
    {For elt <- l1 bound 20 do
        next set = If (iskey? set elt) Then set Else elt:set;
      Finally set
    }
  };
```

Given two lists, this function finds the *union* of them. To do so, it traverses the list *l1* and checks the presence of each of its element in another list *l2*. If it finds an element of *l1* in *l2* then that element is not added to *l2*, otherwise it is added to *l2*.

The body of this function is a block. This block binds *l2* to *set* and computes the union of two lists in a *for-loop* expression. Note that the *for-loop* and binding expressions are evaluated concurrently. The *for-loop* expression has one other statement in its loop body which is *nextified* using the **next** keyword, denoting the new value of *set* to be used in the next iteration, and thus create a dependency among iterations. This loop is bounded to 20.

Each iteration of a loop can be viewed as a separate instantiation of a function. In this example the loop body is like a function with two inputs *elt*, *set* and one output *set*. An iteration computes output *set*, and invokes the next iteration passing *elt* and *set* as the corresponding inputs. When the loop terminates, the final value of the loop is given by the expression following the keyword **next**, which uses the last value computed. In this example, the last *set* is returned.

Non-strictness allows a *for-loop* expression to return the final result, *set*, before all iterations of the *for-loop* finish. Similarly, Id supports *while-loop* expressions. Based on the nature of loops, they can be *Doacross* loops, i.e., the loop iterations may interact with each other, and are synchronized using locks implicitly; or *Doall* loops, i.e., no loop iteration interact with each other, and they can proceed in parallel.

4.4.2 Data Abstractions

Some of Id's functional data types, we used in our implementation, are *lists*, *tuples* and *arrays*. The functional arrays are already described in the previous section.

Lists

For lists, Id has two constructors: the dyadic '*:*' operator to make non-empty lists, and the niladic *Nil* for empty lists. For selection, it uses pattern-matching. Lists can be constructed using the infix operator *:* and its components can be selected by head (*hd*) and tail (*tl*) primitive functions. Consider a function *adj* which takes an

element and a list as its arguments, and produces a list.

```
Def adj e l =
  If (iskey? l e) Then l Else e:l;
```

The presence of an element e in the list l is checked before any arm of the conditional expression is evaluated. This expression returns the same l if e is present, otherwise it concatenates e at the head of l and returns the new list.

Consider the following single-order function *outNodes*. This function constructs a list of *outgoing* nodes from a node *nd* in a graph (denoted by a 2-d *token* matrix) using *List comprehension*. The weight of an edge e_{ij} is represented by *token[i,j]*. If there is no edge between node i and j, then *token[i,j]* contains *maxint*, otherwise the weight of that edge.

```
def outNodes nd n token =
  {: j || j <- 1 to n bound 10 when (token[nd,j] <> maxint)} ;
```

This is read as the list of all indices where j increments from 1 to n. $j \leftarrow$ *1 to n* is called the generator and it is bounded to 10 to limit the degree of unfolding to a fixed number of iterations. This generator is also associated with a filter. When *token[nd,j]* is not equal to MAXINT, then j becomes an element of the list, otherwise it is not.

List comprehension does not add any computational power to the language. However, they are elegant, concise and powerful notations to increase the *expressive power* of the language. We have found that these constructs contribute a significant amount of parallelism if they are used to construct a list of processes (iterations) before we start any loop structure.

Consider a *recursive* function *iskey?*. It performs repetitive computations.

```
def iskey? Nil key = false
 |  iskey? (x:xs) key = (key == x) or (iskey? xs key);
```

This function is defined in two clauses, separated by |. The first clause is used when the list argument is empty, i.e., it matches the pattern *Nil*. The second clause is used when the list argument is non-empty, i.e., it matches the pattern (*x:xs*). In this case, *x* is bound to the head of the list and *xs* to the tail of the list.

In this way, it looks for the existence of an element, *key*, in a given list *xs*. If it is present then it returns true else false. Any call to this function results in a chain of calls that progressively searches the element, *key*, in list *xs*. *Recursion* contributes to the expressive power of Id and exposes parallelism in the code due to non-strictness of the function invocations.

Tuples

Tuples are heterogeneous sequence of values. They are constructed by listing the values separated by commas. Tuple components are selected by pattern matching. Consider the following example:

```
def edgeReplaced? val =
  if (val < maxint) then
    (true,true)
  else
    (false,false);
```

The function *edgeReplaced?* takes one argument that is expected to be an integer[5] *val*. The conditional statement in the function body checks the condition and returns the appropriate arm which is a single two-tuple containing boolean components.

[5]In Id, we do not need to declare the types of all the identifiers used in the program; type inferencing is performed automatically by the system before running the program.

4.5 Non-Functional layer (M-Structures)

Id goes beyond functional languages via the inclusion of I-structures (incremental immutable data structures) and M-structures (mutable data structures).

I-structures are *single-assignment* arrays, hence, they allow a programmer to write *deterministic* programs. It associates synchronization state with every element of an array. An element of this array is either empty or full. When an I-structure array is allocated, all of its elements are *empty.* When an element is written, its state changes to *full* and remains *full* thereafter. Any operation attempting to read an element that has not been written is delayed until the element is written. This synchronization enhances parallelism, since readers and writers can run in parallel, with readers delaying only if they precede the write.

Id programs, restricted to the functional and I-structure layers, are determinate, meaning that consumers are not given a value until it has been produced and producers and consumers interact with each other without affecting the determinacy of a parallel program. There are no side-effects, i.e, memory state does not change during a process's life time. In such an Id program, data structures are updated by re-copying the structure to a new structure instead of modifying the same structure. This results in excessive memory usage and excessive copying. Such Id programs can not work on problems with large program size due to memory limitations of the machine. The *write-once* semantics of I-structures further creates an artificial serialization through *threading* [2]. As a result, *mutable* data structures [2] are introduced without compromising its implicitly parallel, declarative nature.

M-Structures, a set of memory operations, allow reads and writes on the data-structures, and also allow side-effects because such data structures can be updated. Programs can use M-structures to eliminate *threading* and *copying*, and in the process improve efficiency and parallelism. The update to any element of

the M-structure array uses an atomic *read-modify-write* protocol, i.e., once an update begins, no other process sees the value until it is updated. The interaction between concurrent processes that share data are *serialized*, because atomic operations cannot be interleaved. Thus, atomicity eliminates read-write race conditions and ensures that updates are *serialized*[6]. M-structure operations are *implicitly* synchronized for atomicity, i.e., the programmer does not use annotations such as *locks or semaphores* to synchronize these operations. To avoid excessive copying overhead and artificial serialization through *threading*, we restrict our implementation of NDS algorithms to the functional layer and M-structure layer of Id.

Every M-Structure location is either in an *empty* or in a *full* state, in which it contains a value. A *take* operation on an *empty* slot suspends, whereas a *take* on *full* slot returns the value and resets the slot to the *empty* state. A *put* operation is applied only to an *empty* slot. If there are no suspended *take's*, it simply writes the value there and marks it *full*, otherwise the value is communicated to one of the suspended *take's* and the slot remains empty. A *put* operation on a full state is a run-time error.

For example, the following function *initArray* creates a mutable array, with every location containing a value *val*[7].

```
def initArray n val =
  {M_array (1,n) of
   | [i] = val || i <- 1 to n bound 10};

A = initArray 5 0;
```

Consider the statement: $A![j] = A![j] + 1$. In this statement, the *jth* slot of m_array is *taken*, incremented by one, and the result is *put* back. The addition

[6]i.e., atomicity restricts processes to access the shared data serially.
[7]Its type may be integer, float, string, list, or any other datatype!

operator is strict, thus ensuring that the *take* precedes the *put*. The semantics of *take* and *put* guarantees that the cell update is atomic, so if k parallel computations attempt to execute the statement above, *their access to the location will be serialized by the hardware* and the final value of the cell will be its original value plus k.

Explicit Sequencing (Barrier Synchronization)

The following example uses a barrier synchronization:

```
{ Aj = A![j] ;
  ---
  A![j] = 0 ;
In
    Aj }
```

The first statement *takes* the old value out, and the second one *puts* the new value (0) back. The −−− is a *sequentializing barrier*. It ensures that the *take* occurs before *put*. Without it, all bindings in the block are evaluated in parallel, so nothing prevents the *put* operation occurring before *take*. The *sequentializing barrier* allows expressions in the block to be explicitly sequenced. The barrier delays evaluating the expressions below it until all the expressions above it have been completely evaluated.

The functional layer is non-strict by its nature, hence, we do not explicitly encode parallelism and the fine-grain implicit synchronization of M-structures allow us to write our program without worrying about interactions between concurrent processes. However, their use forced us to use explicit barriers to retain determinacy of results[8].

[8]For example, to sequentialize two for-loop expressions which use the same M-array, we put a barrier between them.

4.6 Annotations for Parallelism Control and Resource Management

Id's semantics allow loops to unfold dynamically, constrained only by data dependencies. *I-Structures* and *M-Structures* in Id allow fine-grain producer-consumer parallelism between parent and child procedures and between sibling procedures. In this way, Id exposes too much parallelism and often exhausts machine resources such as frame memory. The amount of resources required are limited by using *bounded* loops, where a bound (a number k) is specified to indicate how many loop iterations can occur in parallel. Moreover, loops can also be specified as *sequential*. The compiler generates specific code for each kind of loop. A *sequential* loop executes one iteration at a time and uses only one frame. A *bounded* loop executes k iterations in parallel, and uses k frames. The sequential loop incurs less overhead than the bounded loop, which has to allocate and initialize more frames. However, it also exploits less parallelism than a bounded loop. Sequential loops are

usually used for the inner-most loops where it is more important to achieve smaller instruction counts than parallelism. Architects of Monsoon suggest that loop parallelism be provided by outer loops. The number of iterations, k, executing in parallel must be large enough to keep the machine busy but small enough not to exceed the storage available on the machine. Categorizing loops as *sequential*, or *bounded* and determining how many bounded loop iterations to execute in parallel is the tedious part of achieving good performance on Monsoon due to the relatively small size of its frame memory.

The second annotation, *@release*, causes heap storage to be reclaimed when it is no longer needed. This allows valuable memory resources to be reused.

4.7 Examples

In this section we discuss four examples to illustrate how some graph problems can be expressed to easily reflect higher level abstractions. This also illustrates how little attention is needed to expose parallelism and how much parallelism is automatically exposed.

4.7.1 Example I: Deleting a node from a list

Consider the following *tail-recursive* function **delNode**. This function deletes *node i* from a list of nodes.

```
def delNode i Nil = error "Element is not found."
 |  delNode i (x:nodes) = if (eq~int x i) then
                             nodes
                          else
                             x:(delNode i nodes);
```

Suppose, we apply *delnode* to some *node i* and non-empty list *Oldlist.*

NewList = delnode i OldList;

As soon as it is verified that the list is non-empty and head **x** is not equal to **i**, the two sub-expressions, *(x)* and *(delNode i nodes)*, may be evaluated in parallel. Let us call the results **z** and **m'**. Non-strict semantics produces and returns the resultant *cons cell (a:b)* immediately, i.e., it does not wait for evaluating **z** and **m'**. It ensures that **z** and **m'** are ultimately stored in the *head* and *tail* of the cons cell that was returned, and any other computation that attempts to read the *head* or *tail* must block (i.e., wait) until the corresponding value appears there. In other words, the producers and consumers of the *head* and *tail* of the cons cell may run in parallel; they simply synchronize at the slots.

4.7.2 Example II: Union of two lists

Consider the following expression.

```
listNew = union (s:nil) (delnode j listOld) ;
```

This expression deletes an element *j* from an old list and then, adds to another list. In *Id*, the call of *union* on *(s:nil)* can begin even though the list produced by the call of *delNode* on its arguments is not available yet. Thus, both calls can proceed in parallel, producing parallelism in our *Id* program. Non-strictness allows the computations of union and delnode to be overlapped. In fact, the *union* operation is performed in a pipelined manner with the *delnode* function, i.e., as soon as the first cons cell (say *l1*) is produced by *delnode*, union may begin its function on both, *s* and *l1*. When *l1* is produced by its expression, the *union* is performed at once.

4.7.3 Example III: Contraction

Fig. 4.5 shows a fragment of our program using M-structures. Given different clusters in the graph, this fragment contracts all the nodes in the clusters to generate a contracted graph represented by only root nodes of the clusters. Its operation is already explained in section 3.3.2. Here, we describe the use of barrier and M-structures for the *producer-consumer* relations in this program.

ActiveNodes is a list of nodes. Each node *i* processes its nodes on the outgoing edges stored in a *CurrentOutNodes* M_array. Each node v_i takes its list of nodes and locks its slot by performing *take* operation on its *CurrentOutNodes* (see the first line in the for-loop body). The *lookup* operation on this slot (denoted by *!!*) in the *if-then* statement is deferred until *NewNodes* are computed by the *computNewNodes* procedure and a *put* is done subsequently. Thus, this process can contribute to producer-consumer parallelism within each iteration. Each node synchronize with others while accessing its root (*s_i*) data structure to contribute producer-consumer parallelism across the loop iterations (*Doacross* loop).

```
{

  %% Each active node i will process its current outnodes and
  %% change each outnode to the root of cluster in which
  %% the outnode is present.

  {for i <- ActiveNodes bound Outer_Loop_Bnd do

     C_Nodes = CurrentOutNodes![i] ;

     % statements deleted.%

   ---

     %% root of cluster.
     s_i = source!![i] ;

     NewNodes = computeNewNodes( C_Nodes );

     CurrentOutNodes![i] = newNodes;

     _ = if (s_i <> i) then
           {
             % statements deleted.%

             Nodes = CurrentOutNodes!![i];

             % statements deleted.%

             CurrentOutNodes![s_i] =
                                union CurrentOutNodes![s_i] Nodes
           }
  }
```

Figure 4.5: A process to contract the graph

4.7.4 Example IV: Constructing a M-matrix

Given *FiringNoArray* and *token* (a 2-d functional matrix), the following function *calculateRToken* constructs a mutable 2-d matrix.

```
def calculateRToken FiringNoArray token =
  { (_,n) = bounds~1D_m_array FiringNoArray ;
    def getVal w f0 f1 =
      { temp = (plus~int w (minus~int f0 F1))
       in
         if (temp < 0) then
           error "Error: Negative Residual Token"
         else
           temp
      }
   In
     {m_matrix ((1,n),(1,n)) of
      | [i,j] = (getVal token[i,j] FiringNoArray!![i] FiringNoArray!![j])
                || i <- 1 to n bound 10;
                   j <- 1 to n bound 10
                 when (token[i,j] <> maxint)
      | [i,j] = token[i,j] || i <- 1 to n bound 10;
                              j <- 1 to n bound 10
                            when (token[i,j] == maxint)
     }
  };
```

The function body is a parallel block. This blocks binds n to the number of nodes in the input graph by computing the bounds of *FiringNo Array* using *bounds 1D_m_array* Id library function. This library function produces a tuple.

Note that '_' denotes the *don't care* value. Next, a function *getVal* is defined and finally, a M-matrix is computed.

We can specify different functions to evaluate the elements in different disjoint regions of the range. The specification of the values in each region follows the special symbol | and the compiler generates one or more loops for each region as needed. All the element bindings can be done in parallel, subject to only data dependencies.

4.8 Summary

In this chapter, we briefly described the Monsoon, Id and Id's parallel execution model. We also discussed the way Id's features help to extract parallelism in the code by using some examples. In the next chapter, we will study the performance of the NDS codes on Monsoon.

Chapter 5

Performance of the NDS Algorithms

In this chapter, we examine the performance of the NDS algorithms on Monsoon, an experimental dataflow machine. To evaluate our code, we have run them on two execution vehicles, namely, a Monsoon Interpreter (MINT) and a real Monsoon machine. We first describe the experimental environment adopted by us to evaluate our Id programs, and then we discuss the experimental results in detail.

5.1 Experimental Environment

Id programs are developed under the Id-World [6] programming environment supported on Monsoon. We used the following Id-world features to collect performance results for the NDS algorithms.

- It can count the number of instructions executed by the machine when it runs the program. The instruction counts can be divided into categories to see what percentage of instructions are remote fetch/store operations versus other kinds of instructions.

- It can classify cycles according to what subalgorithm of our program was being executed, which allows us to see what parts of the program were active at various times, and to see what parts ran in parallel [8]. Fig. 5.1 shows the way we have used their technique for our study. The run time system calls main function which, in turn, calls other subalgorithms. All functions called by a subalgorithm inherits the color of that subalgorithm, e.g., if a function (say *initArray*) is called by two different subalgorithms at different times, then the cycles due to *initArray* will be counted in each subalgorithm's cycles count separately by default.

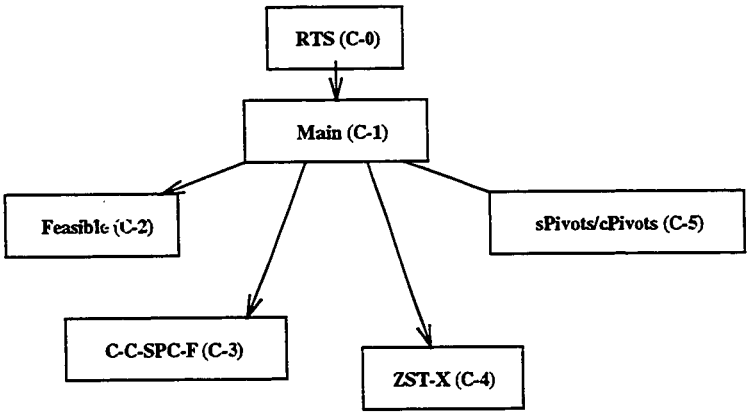

Figure 5.1: Call Tree of NDS algorithm showing coloring and inheritance at work

- Id-World allows us to specify a time interval to customize the statistics resolution. The sampling rate is set according to the length of total computation as well as the resolution required, e.g., to obtain the utilization profile on Monsoon for our code, we set it to 100,000 to keep run-time reasonable. This tells Id-world to sample the instruction count every 100,000 machine cycles. However, to obtain the parallelism profile on MINT for our code, we set it to 100 and it takes about fifteen minutes to obtain the results and profile.

- The run-times are calculated by dividing the total cycle count by 10^7 because Monsoon issues instructions at 10 MHz.

5.1.1 Run-Time-System (RTS)

The RTS [16] is a collection of software procedures linked into every Id program which executes on Monsoon as part of the user's program. The RTS provides four basic services: frame management for procedure activations, heap management for aggregate data, error and exception handling, and I/O.

The I/O routines in the RTS pass requests to the Monsoon interface software running on the host, whereupon the requests are handled and the data value is returned to the RTS for further processing. The RTS overhead to perform I/O, degrades the performance of our Id programs on MINT as well as on Monsoon. So, we have separated this overhead from the overall performance.

The RTS can artificially reduce parallelism by taking a long time to satisfy a frame or heap object request. A processor may need the frame or heap memory in order to continue computation, and thus may idle if that object is not returned by a specific time. A typical RTS request takes 31 or 32 instructions[1] and it executes these instructions sequentially. On Monsoon, the actual latency will be eight times as long because of processor pipeline interleaving. However, these operations take one cycle on MINT.

5.1.2 Monsoon Dataflow Graph Interpreter (MINT)

The dataflow graph produced by the Id compiler can also be executed on MINT [16], an instruction-level interpreter, to produce results and a *parallelism profile*. This profile measures how many operators (instructions) are executed concurrently at each time step (a parallel computation step) in the computation on an ideal machine which has unbounded number of processors and memories, and instantaneous

[1] 16 instructions to read a frame request, choose a processor, and forward a request to that processor; 7 instructions to pop a frame; 8 or 9 instructions to form a context by combining the frame pointer, instruction pointer and statistics color of the called code-block.

communication [11, 6]. Moreover, all operators take unit time, and they are executed as soon as possible. Total number of time steps required to complete a computation is called the *critical path* (an inherently sequential thread in the overall computation).

The parallelism profile measures the *inherent parallelism* of the algorithm because the execution of any two operators is sequentialized if and only if there is a data dependency between them. Hence, it is an ideal reference point against which to measure how much parallelism is actually exploited by a particular machine. This profile conveys the following information.

- The parallelism profile also helps us to detect the inherent sequential portion of an algorithm.
- A single Monsoon processor machine will take at least as many instructions to execute a program as area under the parallelism profile curve, i.e., the total number of instructions executed.

We have evaluated our Id programs along two dimensions:

1. Instruction counts: measures the efficiency of the algorithm as the total amount of work done to compute the solution.
2. Parallelism: average parallelism of the program is calculated by dividing the total instruction count executed in the program by its critical path.

5.1.3 The NDS Algorithms

These algorithms read an N nodes graph in the form of a 1-d vector of node weights, w_i, 2-d vector of edge weights, d_{ij}, from an input data file and solve it in polynomial time. The nature of the input graph greatly affects their performance. To study the performance on MINT, 1pe1is (one processor element and one I-structure unit) and

2pe2is Monsoon systems, we use a graph containing 50 nodes and 500 edges[2]. The structure of this graph is the same as a transportation problem, meaning that there is no transshipment node. It has only producers which produce a commodity, and consumers which consume that commodity.

5.2 Performance of the NDS-CP algorithm

The Id code implementing the NDS-CP algorithm is about 1346 lines long, including comments. Table 5.1 shows the performance of this algorithm on Monsoon (1pe1is). The execution cycles taken by this algorithm are classified according to which subalgorithm of it was being executed. Fig. 5.2 shows a summed total of the instructions executed on 1pe1is.

NDS-CP using ZST-Org		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles $*10^3$	fraction
RTS	13,247	18.90
Main	2,256	3.22
Feasible	27	0.04
C-C-SPC-F	6,573	9.38
ZST-Org	9,588	13.68
cPivots	6,459	9.21
2nd-phase	4,205	6.00
Idles	27,750	39.58
Total	70,103	100.00
Utilization	38,149	54.42
Critical Path	70,103	
Exec. Time	7.0s	

Table 5.1: Breakdown of cycles in the NDS-CP on 1pe1is solving 50 nodes graph

[2]We could not run a graph having more than 50 nodes on MINT. The reason is that MINT is 100 times slower than Monsoon. Moreover, memory limitations further restricted the runs to small input graphs.

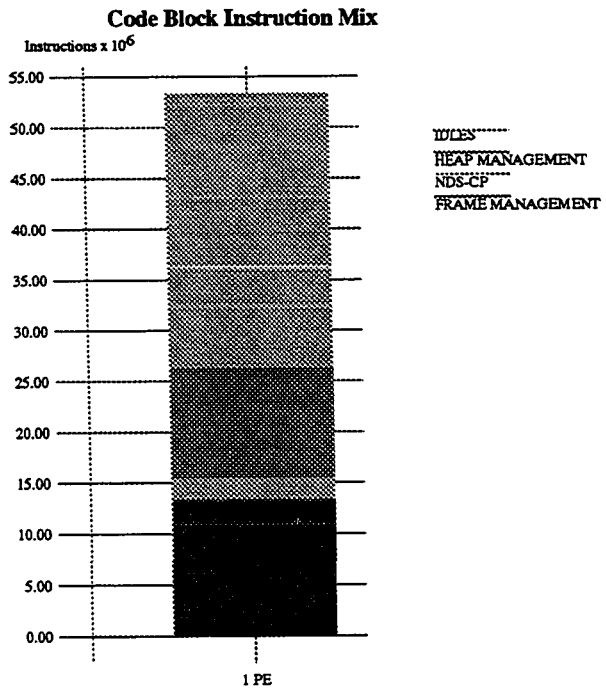

Figure 5.2: Graphical representation of table 5.1

5.2.1 The RTS Overhead

The major contribution to the total cycles is due to RTS operations[3] (18.90%). These operations require a significant amount of parallelism to hide the latency caused by them. Fig. 5.2 shows that the 1pelis spends much more time in RTS than in the compiled code for the NDS-CP.

5.2.2 Subalgorithm Feasible and C-C-SPC-F

The feasibility testing for this 50 nodes graph is not required, so, the contribution of *Feasible* to the total cycles is very small[4]. Hence, the performance study of this subalgorithm is deferred until section 5.6. The first (iterative) step of the optimization phase which finds an initial *bfs* involves Clustering, Contraction, Shortest-path computations and Firing (*C-C-SPC-F*) computations. If there are more than one

[3]These operations are for allocating/deallocating frame and heap memory; and for error and exception handling.

[4]This subalgorithm executes two steps. It initializes the firing numbers during its first step, and then performs testing during the second step. Table shows the contribution due to the first step.

cluster after Clustering, then the remaining computations are also executed, otherwise the NDS-CP algorithm moves to build a 0-token spanning tree. The first iteration of this step works on N nodes. The next iteration works on a contracted graph in which some nodes represent clusters. This process continues until all nodes merge into one cluster. Thus, the number of operations decreases after each iteration, therefore, we see the tremendous variability of the potential parallelism during its execution (fig. 5.6).

Clustering subalgorithm begins with *while-loop* having a nested *for-loop* as its body. The *while-loop* is a sequential loop. Each iteration of this loop forms a phase which is a *Doacross* for-loop. During its first phase (first while-loop iteration), each non-positive node conveys its source information to the node on each of its outgoing zero-edge and all such nodes work in synchrony, i.e., they synchronize with each other while accessing its representative information which is stored in a M-structure array *Source*. The *while-loop* iterates until all nodes (positive/non-positive) finish their job. The *while-loop* is followed by two *for-loop* expressions. During these loops, each node traverses the zero-edge path to correct its representative information. The first one is a *Doall* loop. The second one is a *Doacross* loop. These three loop structures are executed in program order. The Contraction step is a *Doacross* for-loop. All its iterations corresponding to the active nodes work in synchrony. Each iteration contributes to the producer-consumer parallelism while accessing its *ActiveOutNodes* structure as explained in the section 4.7.3. The Shortest-path computation is a *while-loop* expression with a doubly nested *for-loop* expression as its body. The *while-loop* is a sequential loop where each iteration forms a phase. During any phase, each node updates its *distance* and the distance of its adjacent node. Each node works in synchrony with other nodes while accessing its *newDistance* information which is stored in a M_array (see section 3.3.2 in chapter 3). The *while-loop* is repeated until each node reaches its shortest-path distance from its neighboring node. The Firing operation is a doubly nested *for-loop* expression. Each positive-weighted cluster is

fired concurrently during these operations.

5.2.3 The ZST-Org Subalgorithm

After this iterative step, a zero-token spanning tree is built using the *ZST-Org* approach (see section 3.3.3). This subalgorithm works in phases. Each phase corresponds to an iteration of a sequential outer *while-loop*. Each phase executes a *Doacross* for-loop. During a phase, all nodes try to join a shared *tree* structure—an atomic M_list—simultaneously by adding all of the nodes on their adjacent zero-edges. However, the nodes which are already in *tree* or the nodes whose one of the node on their adjacent zero-edges are already in the tree succeed in joining *tree*.

5.2.4 The cPivots Subalgorithm

The next step of this optimization phase is the concurrent pivot operations to improve the objective function, WX. This sequential iterative step involves checking all the edges whose any adjacent node is a leaf node (or cluster) for both types of firing, performing positive firing all the eligible clusters concurrently, negative firing all the eligible clusters concurrently, and then traversing the tree to form new clusters. All these complex procedures are executed in program order. The procedure to check the firing condition for each leaf node/cluster is a *Doacross* for-loop. Each such node examines the only node on its adjacent zero-edge and sets an appropriate flag. The procedure to perform firing involves two iterative steps. The first step calculates the cluster firing numbers and is a *Doacross* for-loop. Each node/cluster interacts with others while updating its representative cluster firing number which is stored in a *Cluster_Firing_No* M_array. The second step performs firing all the clusters concurrently. Finally, the tree is traversed bottom up. It involves forming new clusters which is a *Doall* for-loop (all the variable bindings in this loop-body are executed independently which contributes instruction/variable level parallelism),

and then finding correct representatives for orphan nodes which is a Doacross loop. Both of these loops are executed in program order.

5.2.5 Discussion

The interesting issue here is why there are so many idle cycles in the 1pelis case to start with. To look into this matter, we tried the following two steps and observed:

- When we increase the problem size, we find the same percentage of idles on this configuration, however, absolute number of idles increases.
- We iterated over a fixed set of loop bounds and chose the one that performed best. Still, the Monsoon processor continues to idle for a significant amount. We set the loop bound, k, to 16 for all except inner-most loops, which are sequentialized.

Thus, we believe that either the Id compiler is not able to extract enough parallelism from the subalgorithms to keep the system fully utilized or these subalgorithms lack parallelism.

The parallelism needed to keep 1pelis fully utilized

Monsoon's processor is an 8-stage pipeline. The processor interleaves eight separate threads of computations, where a thread of computation is a sequence of instructions related by control flow that execute successively in the pipeline. Each individual thread of computation actually takes eight times as many cycles as instructions to execute. Therefore, each processor needs *at least eight-fold* parallelism (8 threads to be active) to achieve 100% utilization. It may require more than 8-fold parallelism to hide the latency caused by multi-cycle RTS operations. Architects of Monsoon experience that it is about 15/20-fold parallelism which is required to keep 1pelis fully utilized.

The utilization profile

The utilization profile shown in fig. 5.3 reveals how the 1pelis is utilized over the course of the NDS-CP. The processor executes two iterations—both are separated

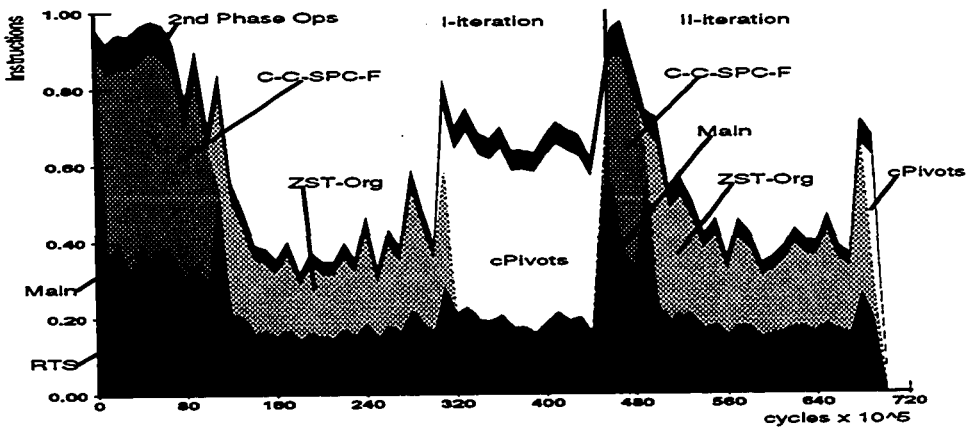

Figure 5.3: Utilization profile of NDS-CP with ZST-Org solving 50 nodes graph on 1pelis

by a black vertical line in the profile—for it to solve 50 nodes graph. During the first iteration, the observations are as follows:

1. The processor is utilized about 90% of its time while finding a *bfs*[5].
2. The deep valleys corresponds to tree-building subalgorithm (*ZST-Org*). The processor utilization is decreased to about 40% while constructing a dual feasible tree.
3. The utilization is further improved to about 65% while finding an optimal solution using *cPivots*.

Similar behavior can be seen during the second iteration.

[5]To find a bfs, NDS-CP executes Clustering, Contraction and Shortest Path Computations & Firing (*C-C-SPC-F*) subalgorithms iteratively.

Observation

Thus, it is due to poor performance of the tree-building subalgorithm (*ZST-Org*) which reduces the overall average processor utilization of NDS-CP to only 54%. Moreover, table 5.1 and the corresponding profile 5.3 reveals that this algorithm [4] spends most of its time in constructing the 0-token spanning tree among all sub-algorithms. The reason is that all N active processes (each one corresponding to a node in the N nodes graph) try to access a shared structure (*tree*) simultaneously and many of them, when they get access to it, end up with doing nothing because they do not find their adjacent nodes in *tree*. The nature of this method to build the tree was causing it to take a lot of time and contribute a lot of *idles*. This motivated us to re-write this subalgorithm to improve the overall performance of the NDS-CP. We designed two new subalgorithms to achieve our goal, namely, the zero-token spanning tree using sequential merging (*ZST-SM*) and the zero-token spanning tree using hierarchical merging (*ZST-HM*). Note that the pivoting steps are directly dependent on the structure of a spanning tree, which in turn, is dependent on the tree-building subalgorithm. As a result, the number of cycles taken by the pivoting steps were also effected.

The ZST-SM Approach

In this approach, we perform two passes. During the first pass, we start a process for each node of the graph. During this process, each node v_i performs the following steps.

1. Collect all adjacent nodes connected to it with outgoing zero-edges to form a subtree.

2. Store node v_i as a root to be active in the second step if it forms a subtree of at least one zero-edge.

All N processes work in synchrony because each node informs all other nodes on its adjacent zero-edges about its presence in a subtree (see section 3.6.1) during first step and a 0-token spanning tree is built which may have cycles. We remove these cycles during the second stage. The second pass works in phases. During each phase, we start a process for each root node v_i found during the previous pass. Each subtree rooted at node v_i joins the main tree either if it is the first one to do so or if it or one of its adjacent nodes is already in the tree. But if it has more than one node already in the tree, then it deletes all but the lowest numbered node from the tree. By doing so, cycles are removed and we obtain the required spanning tree. This pass is inherently sequential.

NDS-CP with ZST-SM		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles $*10^3$	fraction
RTS	11,309	19.66
Main	2,268	3.94
Feasible	27	0.05
C-C-SPC-F	6,777	11.78
ZST-SM	4,638	8.06
cPivots	9,521	16.55
2nd-phase	3,676	6.39
Idles	19,310	33.57
Total	57,526	100.00
Utilization	34,541	60.04
Critical Path	57,526	
Exec. Time	5.7s	

Table 5.2: Breakdown of cycles in the NDS-CP using ZST-SM on 1pe1is

With *ZST-SM* method (see table 5.2), we could achieve our purpose of reducing the number of *idles* and the number of cycles to construct the tree. However, the number of cycles takes by *cPivots* increased significantly (cf. table 5.1). The number of RTS operations using this approach is reduced which results in more parallelism

because the processor spends less time in sequential RTS. The decrease in number of *idles* increased the processor utilization. The NDS-CP algorithm executes two iterations of the optimization phase to optimize this graph in 5.7 sec.

The ZST-HM Approach

To circumvent the sequentiality in the merging, we can allow the subgraphs to merge with its neighbors in a hierarchical fashion. For example, suppose we have subgraphs represented by v_1, v_2, v_3 and v_4, we can merge v_1 and v_2 at the same time that we merge v_3 and v_4. If they merge, then two resulting larger subgraphs represented by v_1 and v_3 can be finally merged to form the tree. However, if only one of them merge (say first pair), then resulting larger subgraph represented by v_1 can be merged with v_3, and then with v_4. If none of them merge, we perturb the subgraphs, and then v_1 and v_3 can be merged at the same time as v_2 and v_4 to form the tree and it goes on until all four subgraphs are merged into one.

In each merging phase of the *ZST-HM* subalgorithm, all subtrees form a pair (it is, more or less, a sequential task), and then each pair tries to merge and removes a cycle if it has one concurrently. After each phase, the number of subtrees decreases by half if all pairs succeed in merging. If only some of the pairs succeed in merging, then those pairs which do not succeed cause unavoidable overhead of spawning useless processes. At this point, the new pairs are formed which will be less than the previous phase and the hierarchical merging is re-started. Furthermore, if no pairs succeed in merging, then we need to permute the subtrees in the hopes of forming pairs which may be merged. This permutation greatly affects the performance of this approach because, in the worst case, if only a few pairs (say only one) succeed in merging in each phase, then this process of merging becomes sequential.

With *ZST-HM* method (see table 5.3, fig. 5.5 in bargraph format and the utilization profile 5.4 on 1pelis), the number of idles are further reduced, hence, the processor utilization is further increased. The number of RTS operations using

NDS-CP with ZST-HM		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles $*10^3$	fraction
RTS	11,161	21.80
Main	3,320	6.48
Feasible	27	0.05
C-C-SPC-F	6,943	13.56
ZST-HM	7,095	13.86
cPivots	6,170	12.05
2nd-phase	3,742	7.31
Idles	12,745	24.89
Total	51,202	100.00
Utilization	34,715	67.80
Critical Path	51,202	
Exec. Time	5.1s	

Table 5.3: Breakdown of cycles in the NDS-CP using ZST-HM on 1pelis

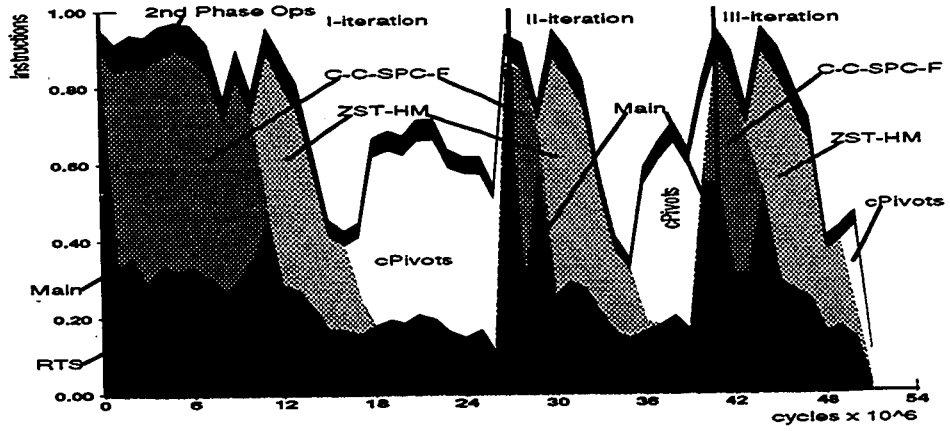

Figure 5.4: Utilization profile of the NDS-CP with ZST-HM solving 50 nodes graph on 1pelis

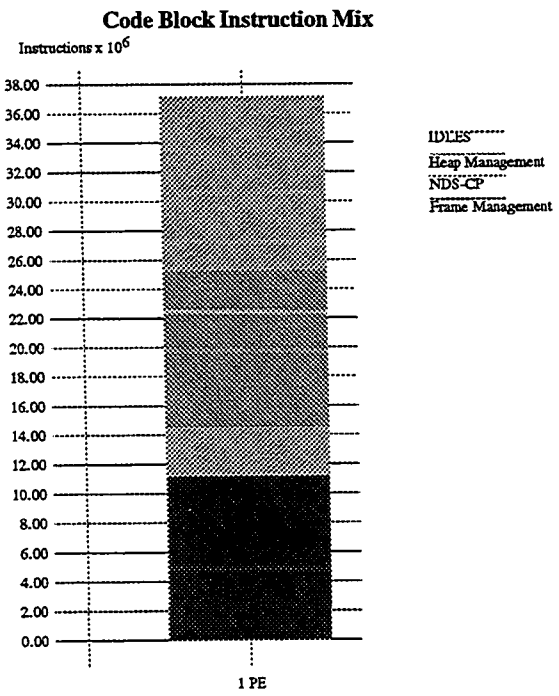

Figure 5.5: Graphical representation of table 5.3

ZST-HM approach is further reduced. The number of cycles taken by this method to construct the tree are less than *ZST-Org* approach. Moreover, the cycles taken by *cPivots* are also minimum. Thus, the 0-token spanning tree constructed by *ZST-HM* method allowed more pivot steps to be done concurrently. Furthermore, the NDS-CP algorithm executes three iterations of the optimization phase to optimize this graph in 5.1 sec. The reason is that the pair-formation step during this method is the unavoidable sequential step which makes it take more time to build the tree. Moreover, the permutaion of subtrees after any phase (if needed) may increase the time to build the tree even though the permutation is done concurrently.

Though, both new methods to build the tree improved the overall performance of the NDS-CP algorithm, the 1pelis is still idle for 25% of the time. Next, we study the parallelism in each of these tree-building approaches to select an appropriate approach for our further study.

5.2.6 Parallelism in Tree-Building Subalgorithms

To study the effect of the three tree-building approaches on overall parallelism, we allow MINT to calculate the initial basic feasible solution (*bfs*) of NDS algorithms, and then allow it to build the tree using three approaches one by one. In this way, we can study the effect of each on the parallelism in *finding an initial bfs.*

Parallelism in finding an initial bfs (C-C-SPC-F)

Table 5.4 and its graphical representation in profile 5.6 show the parallelism in finding the initial *bfs.* The average parallelism of 52 shows that these computations have enough independent operations to hide latency caused by the multi-cycle RTS operations at each step during their execution. Hence, the Monsoon processor will be utilized efficiently during these computations. Even though most of the loops

Finding an initial bfs of NDS algorithm		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles $x10^3$	fraction
RTS	3,094	31.21
Main	794	8.01
Feasible	26	0.26
C-C-SPC-F	5,148	51.92
Second Phase	853	8.61
Total	9,916	100.00
Critical Path	191	
Parallelism	52	

Table 5.4: Breakdown of cycles in finding an initial *bfs*

in this iterative step are *Doacross* in nature (see section 5.2.2), these computations contribute the maximum amount of parallelism to the overall parallelism of this algorithm. These loop iterations synchronize with each other using implicit locks associated with each element of the M-structure. This shows that the fine-grain (element-level) synchronization using such locks helps processes to interact more

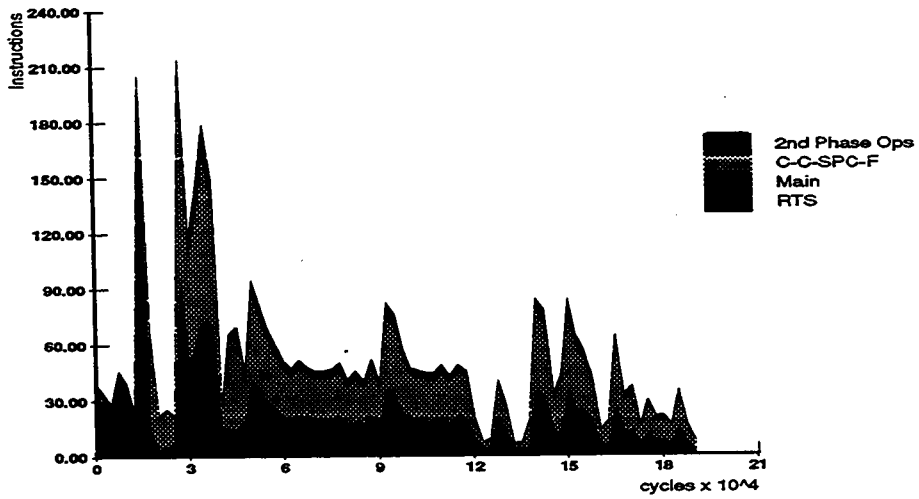

Figure 5.6: Parallelism in finding an initial *bfs* of 50 nodes graph

freely because they can operate on different elements of the same data structure simultaneously. However, if any process need to use some element which is being updated by some other process, then it waits until the latter process is done with it.

Parallelism in ZST-Org

The *ZST-Org* approach is sequential in nature because a node joins *tree*—it is an atomic variable—one at a time. This fact is visible in its profile 5.7. Using this approach (table 5.5), the RTS operations are increased by 3317 and parallelism is reduced by 30 (cf. table 5.6). Obviously, the hardware will be idle because the more RTS involved, the more parallelism is needed to be exploited by the hardware for its 100% utilization.

Parallelism in ZST-SM

Using *ZST-SM* approach (see fig. 5.6), the RTS operations are increased by 1760 and the parallelism is reduced by 21 (cf. table 5.6). However, it performs better than *ZST-Org* because the contribution of the RTS and the cycles to build the tree

NDS and ZST-Org		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles $x10^3$	fraction
RTS	6,411	33.55
Main	1,154	6.04
Feasible	26	0.13
C-C-SPC-F	5,148	26.95
ZST-Org	4,709	24.65
Second Phase	1,658	8.68
Total	19,106	100.00
Critical Path	891	
Parallelism	22	

Table 5.5: Effect of ZST-Org on *bfs*

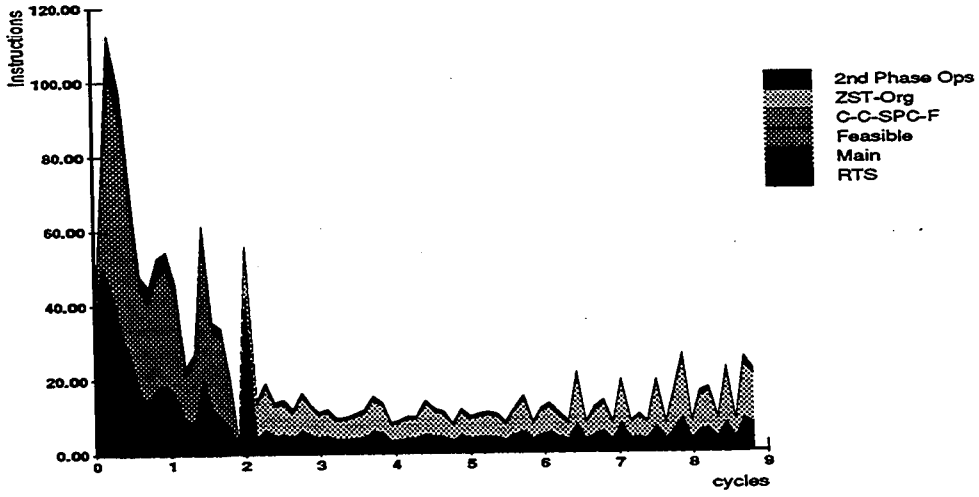

Figure 5.7: Parallelism in ZST-Org

are less. It shows that the hardware will be idle, however, not as much as in the case of *ZST-Org.* The merging portion of the *ZST-SM* subalgorithm is sequential in nature. However, it performs better because of the parallelism in its subtree-building portion.

NDS and ZST-SM		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles $x10^3$	fraction
RTS	4,854	32.84
Main	1,154	7.80
Feasible	26	0.17
C-C-SPC-F	5,148	34.83
ZST-SM	2,322	15.70
Second Phase	1,280	8.66
Total	14,783	100.00
Critical Path	480	
Parallelism	31	

Table 5.6: Effect of ZST-SM on *bfs*

Parallelism in ZST-HM

Using *ZST-HM* approach (see table 5.7 and profile 5.8), the RTS operations are increased by only 1460 and parallelism is reduced by only 12 (cf. table 5.6). The *ZST-HM* approach performed better than the earlier two methods because the contribution of the RTS is minimum. As a result, the processor will spend more time in the compiled (dataflow-style) code than in the sequential RTS code and it will be utilized more efficiently. We observe (see profile 5.8) that the beginning phases of the hierarchical merging find enough pairs to merge concurrently which may result in a significant amount of the loop-parallelism in this critical part of the NDS algorithms.

With this study, we choose the *ZST-HM* approach to build 0-token spanning

NDS and ZST-HM		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles $x10^3$	fraction
RTS	4,554	31.47
Main	1,154	7.97
Feasible	26	0.18
C-C-SPC-F	5,148	35.58
ZST-HM	2,352	16.26
Second Phase	1,238	8.55
Total	14,471	100.00
Critical Path	365	
Parallelism	40	

Table 5.7: Effect of ZST-HM on *bfs*

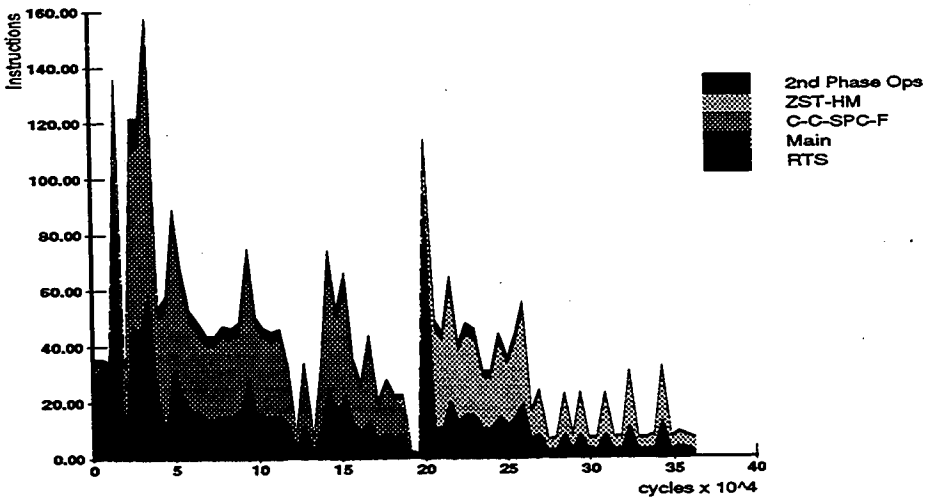

Figure 5.8: Parallelism in ZST-HM

tree for our further study of the NDS algorithms.

5.2.7 Parallelism Study of NDS-CP

The results presented in the previous section motivated us to investigate the amount of parallelism in the concurrent pivoting used in the NDS-CP. To do so, we run this algorithm on MINT to solve the same problem using the *ZST-HM* method to build the tree. The parallelism profile of the first outer-loop iteration of the optimization phase is shown in fig. 5.9. This profile identifies some important structural aspects of the parallelism. The parallelism during the *C-C-SPC-F* computations varies from

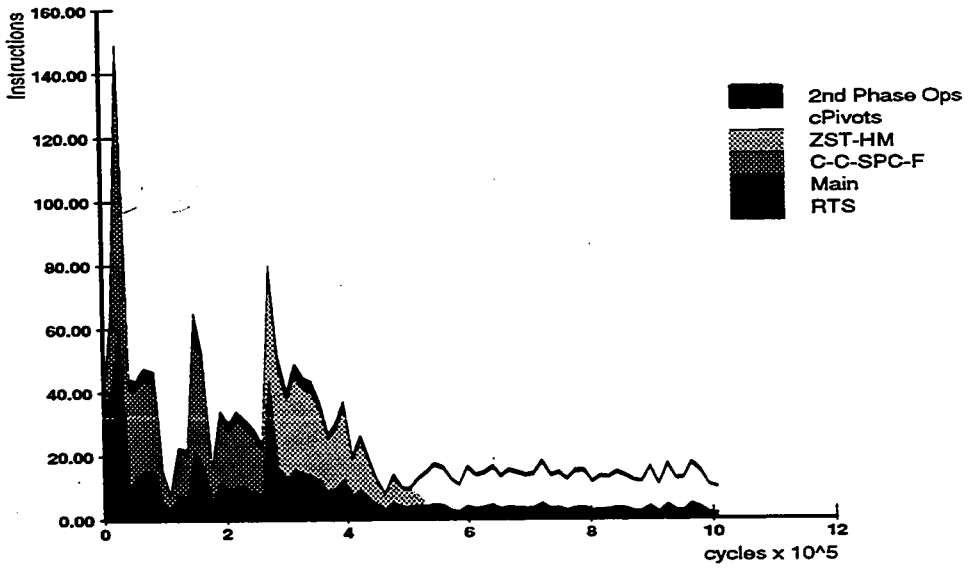

Figure 5.9: Parallelism profile of first iteration of the NDS-CP solving 50 nodes graph

about 80 to 35 operations[6] as shown in profile 5.9. The continuous decrease in the potential parallelism (from about 35 to about 8 or 9 operations) due to hierarchical merging is visible during its execution. Its profile 5.9 indicates that there is a little but constant parallelism (12 to 15 operations) in the concurrent pivot steps.

[6]The average is about 52 independent operations as discussed in the previous section.

After the concurrent pivoting, the NDS-CP algorithm moves to the first step of optimization phase. In this way, the NDS-CP executes three iterations of this phase to find an optimal solution. The first iteration of the optimization phase contributes an average parallelism of 25. However, it decreases during the second and third iterations. The overall average parallelism of the NDS-CP is approximately 20.

This profile explains that the overall parallelism is diminished due to the lack of parallelism primarily in the concurrent pivot operations. As a result, 1pe1is is idle 25% of the total cycles. The average utilization is 68%. The reason is that these pivoting operations interact with RTS heavily (see in any table) for allocating/deallocating heap storage, activation frames. These operations take one cycle on MINT, however, they require multiple cycles on the real hardware. Moreover, there must be enough amount of operations which are not dependent on them to keep the processor pipeline busy. However, the profile shows that there are not enough operations (or parallelism) to mask the latency caused by these multi-cycle operations during *cPivots*. As a result, we find a huge amount of the idles (see table 5.3).

5.2.8 Performance on 2pe2is

For the same reasons discussed above, the utilization on a 2pe2is Monsoon decreases to 45% even though the operations executed by both configurations for each subalgorithm are the same (cf. table 5.3). The reasons for more *idles* on 2pe2is are as follows:

- A subset of *Tag* operations send a token to the network and cause a hardware hazard. These operations can not cause *idles* on 1pe1is, however, they do on a multiprocessor Monsoon because the callee procedure can be on another processor (see section 5.3.3).

NDS-CP		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles$*10^3$	fraction
RTS	11,166	14.73
Main	3,341	4.41
Feasible	26	0.03
C-C-SPC-F	6,956	9.18
ZST-HM	7,038	9.29
cPivots	6,077	8.02
2nd-Phase	3,625	4.78
Idles	37,570	49.57
Total	75,799	100.00
Utilization	34,604	45.65
Critical Path	37,900	
SpeedUp	1.35	
Exec. Time	3.7sec	

Table 5.8: Breakdown of cycles in the NDS-CP on 2pe2is solving 50 nodes graph

- The execution of Id on Monsoon starts with the invocation of the top level procedure on one processor and spreads out to other processors. The computation ends in a similar way, but in reverse order on the same processor that the computation started on. Startup and end costs[7] are incurred whenever the architecture and/or the RTS limits the expansion or contraction of work to/from the processors. When the 1pelis shows 1% idles due to startup and end costs, an n-processor system will generally show at least n% accumulated idles due to startup and end costs.

- Too little parallelism, which results in the lack of work, is much more likely on a multiprocessor since more parallelism is required to keep it busy. Since the Monsoon processor consists of an eight-stage interleaved pipeline, a 2-processor Monsoon requires at least 16-fold parallelism to achieve 100% utilization. It

[7]These costs are visible as ramp up and ramp down in the utilization profile.

may require more parallelism to hide the latency caused by the RTS' multi-cycle operations.

5.3 Performance of the NDS-SP Algorithm

The Id code implementing the NDS-SP algorithm is about 1202 lines long, including comments. Table 5.9 shows its profile classified by each subalgorithm. This profile shows a sequential flow (shown in fig. 3.5) on the same 50 nodes graph. In this

NDS-SP		
Problem Size, N = 50 M = 500		
Subalgorithm	cycles$*10^3$	fraction
RTS	7,554	27.21
Main	1,154	4.16
Feasible	26	0.09
C-C-SPC-F	5,148	18.54
ZST-HM	2,352	8.47
sPivot	9,401	33.86
Second Phase	2,128	7.66
Total	27,763	100.00
Utilization	25,635	92.34
Critical Path	1,141	
Avg. Parallelism	24.33	

Table 5.9: Breakdown of cycles in the NDS-SP solving 50 nodes graph

case, the *C-C-SPC-F* computations took about 19% of the total cycles to find a *bfs* of the given graph. The tremendous variability of the potential parallelism during its execution is due to the continuous decrease in the number of operations (see section 5.2.6). The improvement of the potential parallelism, and then further decrease is visible during the *ZST-HM*'s execution. Thus, the average parallelism is about 39 operations (see table 5.7 in section 5.2.6) before the execution of *sPivots*.

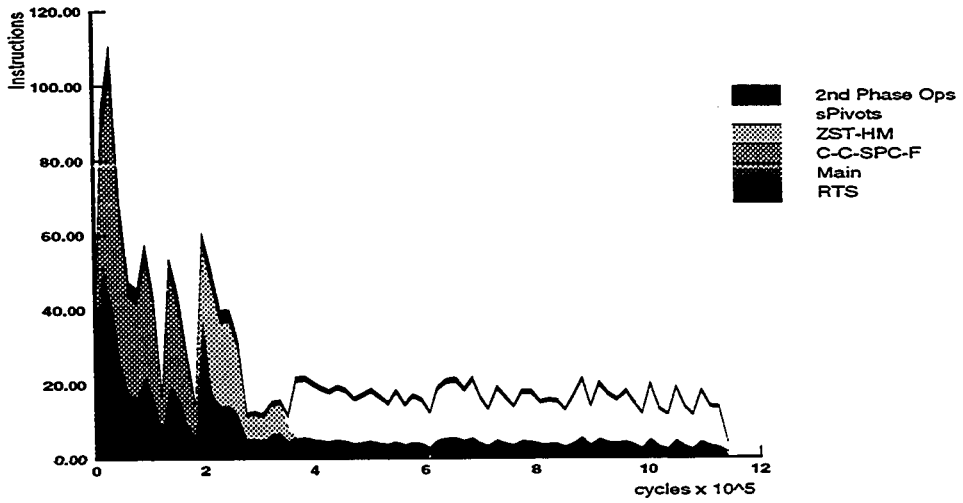

Figure 5.10: Parallelism profile of NDS-SP solving 50 nodes graph

5.3.1 Parallelism in the Sequential Pivoting

The next step of the NDS-SP is to optimize the *bfs* using sequential pivoting. It executes 476 iterations (pivot steps) to find the optimal solution. During each such step, it checks a zero-edge for the firing condition. If it satisfies the condition, then a node/cluster is fired. Firing a node/cluster involves two iterative steps. During the first step, it calculates the firing number (see section 3.2) and all the nodes in the cluster are fired concurrently. After this operation, if firing was done then it modifies the tree and restarts the next phase of pivoting steps. Otherwise, the tree is traversed from this edge to form the new cluster. All the statements before the for-loop expression (see fig. 3.15 in chapter 3) in traversing the tree are executed independently which contributes instruction (variable) level parallelism. The next phase of pivoting steps cannot start unless the tree is modified by the previous phase. All sub-steps of a pivoting step are executed in program order.

Observations

- The *sPivots* took about 34% of the total cycles to optimize the initial *bfs* because it processes one zero-edge at a time.
- About 3,000,000 cycles are due to RTS during *sPivots*' execution (cf. table 5.7). However, these pivoting operations do not have enough parallelism to hide the latency caused by these RTS operations which take single cycle on the MINT but multi-cycles on a real Monsoon.
- The overall average parallelism in the NDS-SP algorithm is further reduced from 39 to 25 operations after *sPivots*' execution.
- A total 28,616,000 operations were executed in overall execution of the NDS-SP. However, the critical path requires only 1,163,000 clocks. This indicates that a single Monsoon processor would execute more than 28,616,000 operations (area of the parallelism profile) to optimize the solution due to the multi-cycle RTS operations which are 30% of the total execution time.

5.3.2 Performance of the NDS-SP on 1pe1is and 2pe2is

Table 5.10 shows the dynamic distribution of operations executed by Monsoon according to what procedure of the NDS-SP algorithm was being executed and fig. 5.11 refers to the utilization profile obtained on *1pe1is* during this execution.

Observations

- Almost the same number of operations were executed as predicted by MINT during each subalgorithm (cf. table 5.9). However, there is a tremendous amount of *idles* on both configurations. The *ZST-HM* causes 2,405,000 idles[8]

[8]It is impossible to calculate exact number of idles caused by each part in any algorithm on Monsoon. However, the nature of this algorithm helps us to determine approximately the number of idles caused by each subalgorithm. We allowed Monsoon to generate the *bfs*, and then we looked

Performance of NDS-SP on Monsoon				
Problem Size N = 50				
Configuration	1pe1is		2pe2is	
Subalgorithm	cycles$*10^3$	fraction	cycles$*10^3$	fraction
RTS	7,533	19.21	7,535	11.60
Main	1,158	2.95	1,156	1.78
Feasible	26	0.07	26	0.04
C-C-SPC-F	5,119	13.06	5,120	7.88
ZST-HM	2,350	5.99	2,348	3.62
sPivots	9,393	23.95	9,393	14.46
Second Phase	2,509	6.40	2,414	3.72
Idles	11,126	28.37	36,956	56.90
Total	39,214	100.00	64,947	100.00
Utilization	25,579	65.23	25,578	39.38
Critical Path	39,214		32,474	
SpeedUp	1		1.2	
Exec. Time	3.9 sec		3.2sec	

Table 5.10: Breakdown of cycles in NDS-SP on Monsoon

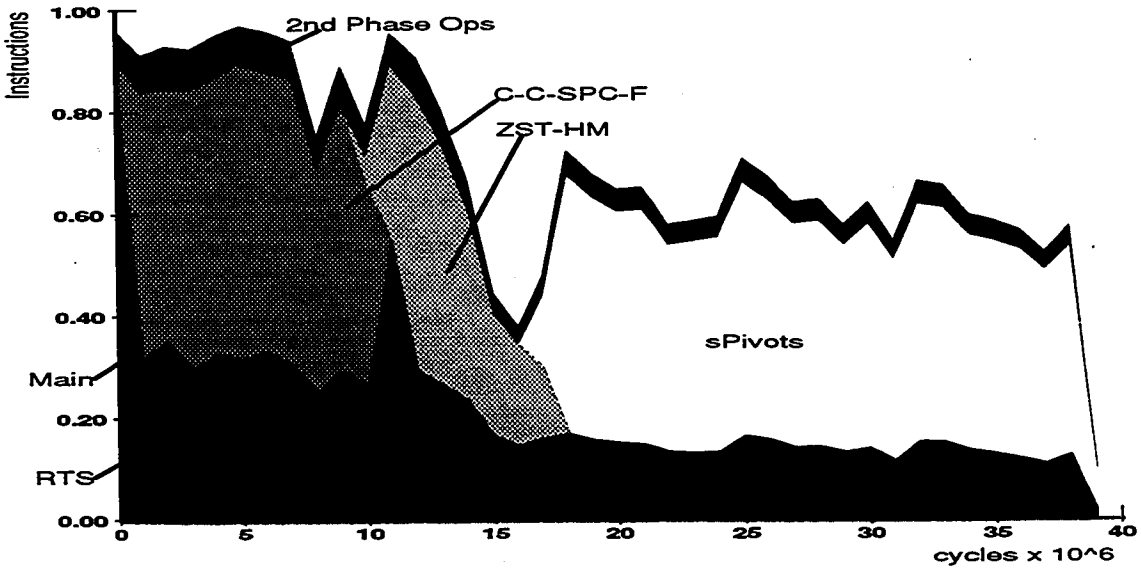

Figure 5.11: Utilization profile of NDS-SP solving 50 nodes graph on 1pe1is

, the sequential pivot (*sPivot*) results in 7,824,000 idles and the remaining 691,000 are due to rest of the sub-algorithms on 1pe1is.

- The 1pe1is is utilized well during the execution of *C-C-SPC-F* and *ZST-HM* subalgorithms. The deep valley in this utilization profile is due to the last few phases of the hierarchical merging of *ZST-HM*. Then, the utilization improves to 67% during *sPivots*. The overall average utilization is about 67% on 1pe1is configuration.

- The available parallelism of 25 is enough only to keep 1pe1is 67% utilized. The utilization on the 2pe2is is further decreased to 40% and the 2pe2is is idle for 57% of the total time.

- The 1pe1is took 3.9sec to solve this graph using NDS-SP. However, it took 5.1sec to solve the same graph using NDS-CP. The same behavior can be seen on 2pe2is.

5.3.3 Quantifying Overheads on 1pe1is

Table 5.11 shows the profile classified by operation count. *Move* operations (21.06% of total cycles) move data in and out of the frame memory on Monsoon processor, i.e., they are basically frame fetches and stores.

Bubbles are caused by the arrival of the first token of a dyadic operation. The instruction can't execute until the second token arrives, so the first value is stored into the frame and the rest of the pipeline does nothing for this token. Bubble instructions (12.74 % of total cycles) in Monsoon is an overhead of the dataflow-style of fine-grain parallel execution supported on Monsoon. *Misc* operations (8.60 % of total cycles) operations consists of control-flow, data conversion, control register

at the profile to see the number of idles. Next, we allowed it to build the tree using *ZST-HM* and observed the increase in the number of idles. This gave us the approximate value of the number of idles due to this tree-building subalgorithm.

Opcode Mix							
Algo/Op	RTS	NDS-SP	Feasible	C-C-SPC-F	ZST-HM	sPivots	%
Bubbles	66	314	6	1,356	608	2,646	12.74
Move	1,552	417	9	1,908	868	3,363	20.70
Int	1,540	137	3	439	195	981	8.40
Fetch	1,224	54	2	172	75	333	4.75
Store	326	24	0	106	49	249	1.93
Tag	1,821	81	1	462	236	580	8.11
Misc	1,004	131	4	676	316	1,242	8.60

Table 5.11: Opcode Mix of NDS-SP solving 50 nodes graph 1pelis

operations. *Tag* operations (8.11 % of total cycles) are the operations that manipulate continuations, such as sending arguments to and receiving results from a called procedure.

Fetch and *Store* operations on Monsoon are used for accesses to I-structure units. Some of the *fetches* which occur in dataflow-style issue split-phase remote memory requests and cause a hardware hazard. About 4.75% of the total cycles executed are *fetch* operations. Table 5.11 shows that the RTS which uses threaded-style code[9] executes more fetches than our subalgorithms which uses dataflow-style code.

Observations

- We estimate the number of idles on 1pelis to roughly equal to the number of fetches that will go to I-structure unit times 1.25, i.e., (54 + 2 + 172 + 75 + 333 = 637) x 1000 x 1.5 = 796,250. Hence, out of 11,126,000 idles (see table 5.10), 796,250 idles (8%) are due to the Monsoon hardware hazard and rest of them (10,329,750) are caused by the lack of work in the algorithm itself.

[9]Dataflow-style code consists of tokens flowing along the arcs of a dataflow graph, whereas, the threaded-style code treats a linear sequence of dataflow instructions as a sequential thread. The Id compiler generates dataflow-style code for Monsoon. Nevertheless, it is also possible to write threaded-style code using assembly language for Monsoon.

- *Bubbles*, *Tags*, *Misc* are all operations that are identified as the overheads of asynchronous execution. The total number of cycles (about 30% of total cycles) spent in these categories gives us a quantitative indication of the cost of asynchronous execution. So, the overhead incurred by parallel execution is reasonably high, increasing total run-time.

- The useful computations for this code on Monsoon (1pe1is) are only 20,583,000 operations[10] though the total number of operations executed are 39,214,000. Thus, the style of dataflow execution on Monsoon introduce huge overheads.

5.4 NDS-SP versus NDS-CP

Our results show that NDS-SP algorithm spends most of its time in pivoting steps. Thus, any parallel implementation of NDS-SP has to focus on the parallelization of the pivot steps. In NDS-CP [4], they parallelized this method by performing concurrent pivot steps. However, after concurrent pivot steps, the solution loses its basisity, while it is maintained by sequential pivot steps in the NDS-SP algorithm. As a result, NDS-CP algorithm goes back to retain its basisity as shown in fig. 3.23. Our detail analysis results show that by doing so, NDS-CP algorithm spends more time in reconstructing the 0-token spanning tree than performing other operations. Through our parallelism study, we found that the dataflow-style implementation for *cPivots* did not have enough parallelism to be exploited by Monsoon processor. As a result, the NDS-SP which has a little more parallelism performed better than the NDS-CP for 50 nodes graph.

Fig. 5.12 plots the critical path length versus different problem sizes[11] for both

[10] 25,578,000 (utilization) - 4,996,000 (bubbles) = 20,583,000. Utilization includes all operations except 2nd-phase and idles.

[11] The problem size is the number of nodes in the input graph because all the computations involve modifying the attributes of graph nodes (e.g. source information of node, weight of a node, firing number of a node or a cluster). Though the pivoting operations also involve operations on edge's residual tokens, these operations are again dependent on adjacent node's attribute (firing

algorithms. The results on this graph are obtained on 1pe1is configuration. The

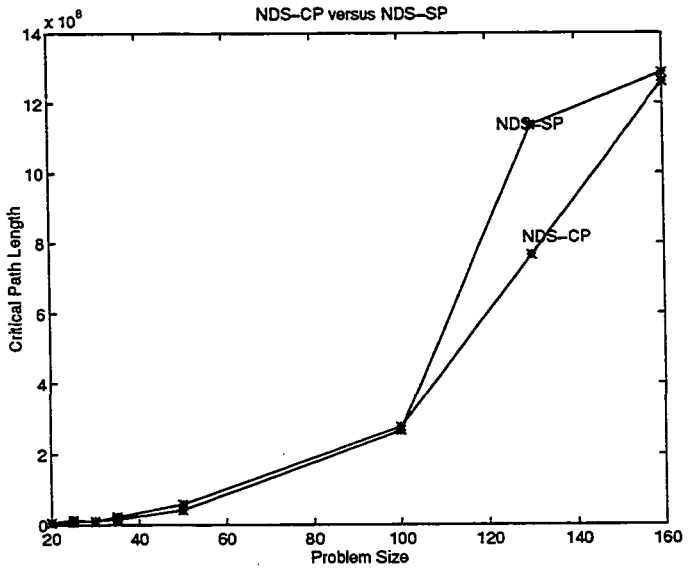

Figure 5.12: Critical path versus Problem size

NDS-SP algorithm took less time to solve the DTP of small problem sizes than the NDS-CP algorithm. It can be concluded that the smaller DTP's do not have enough parallelism—because of less pivot steps which can be done concurrently—to exploit, so the parallel approach (the NDS-CP algorithm) to solve them took longer than the sequential approach (the NDS-SP algorithm). However, the NDS-CP solves 130 nodes and 2500 edges graph faster than the NDS-SP. The reason is that the *sPivots* spend 42% of the total time to find an optimal solution and the *cPivots* spend only 19% of the total time to find the same solution. Moreover, the NDS-CP executes three iterations of the optimization phase for this graph in 76 seconds and the NDS-SP finds the same solution in 108 seconds. Thus, this graph do have enough parallelism—because more pivot steps can be done concurrently—to be exploited and hence, the NDS-CP solves it faster than the NDS-SP algorithm on Monsoon. For 160 nodes and 3500 edges graph, the NDS-SP again leads the NDS-CP.

As already discussed in the previous sections, these algorithms are input graph

number.).

dependent. If a graph allows more pivot steps to be done concurrently, then NDS-CP leads the NDS-SP.

5.5 Id's features for parallelism in the NDS algorithms

We tried to investigate the contribution of each Id feature, used in our implementation, to the overall parallelism. To do so, we first sequentialized the NDS-SP parallel code. All loops, *Doall_For* and *Doacross_For*, are sequentialized by using the annotation **sequential** and their loop-bodies are sequentialized using *barriers*. All statements in a block are also forced to run sequentially using *barriers*. This approach made all functions execute in a *strict* manner (i.e., all the arguments are evaluated before any computation of the function body starts). However, we could not achieve parallelism of one from our sequential code. This is due to instruction-level parallelism and the parallelism in the RTS, which shows up when an Id program is run under idealized mode. We consider a parallelism of 3.83 as our baseline and allow one construct at a time to contribute its parallelism to overall parallelism. Fig. 5.12 shows this contribution.

There are a few *Doall_For* loops, so their contribution to overall parallelism is not much. However, our program uses mostly *Doacross_For* and *Doacross_while* loops, so their contribution is quite high. *Doacross_while* loops form the outer loops and *Doacross_For* loops form the inner loops in most subalgorithms. These loop iterations synchronize using implicit m-locks associated with each element of a shared data structure. These implicit atomic data structures allow the parallel processes to interact more freely. After this, we removed the *barriers* with in the *loop-bodies* and allowed them to run in a *dataflow-style*. These loop-bodies behave like a parallel block after removing barriers. Their contribution is very little because of the extensive *data-dependencies* between instructions.

The NDS-SP Algorithm	
Problem Size, N = 50	
Feature Name	Parallelism
Doall_For	0.51
Doacross_For	4.25
List comprehension	6.12
Array comprehension	2.55
Doacross_While	2.17
Parallel block	2.03
Total	17.63

Table 5.12: Id's features for parallelism

The maximum amount of contribution comes from the use of *List comprehensions*. We used this structure to select a subset of items (processes to be forked in a loop expression) generated by it.

Total parallelism due to different features is 17.63. We started with the initial parallelism of 3.83. Thus, overall parallelism is about 22. This overall parallelism is further improved after removing *barriers* from other portions of the code, which allowed *functions* to evaluate their arguments in parallel with their function body. Thus, overall parallelism in our NDS-SP code is 25.

5.6 Solving ILCWB Problem using NDS-SP and NDS-CP

In this section, we study the performance of the NDS algorithms solving a ILCWB problem formulated as 149 nodes DTP on Monsoon/MINT. To maintain the compacted width of the layout after compaction, an artificial edge is added to the constraint graph after the application of Feasible (see section 3.5).

5.6.1 Performance of 149 nodes ILCWB problem on 1pe1is

To set the loop bounds, we iterated over a fixed set of loop bounds, k (20, 30, 35 and 40) and chose the loop bound, k=30, that performed best. For loop bound, k=40, 1pe1is showed the error *out of frame memory* and for loop bound, k=35, it performed poor than for the case, k=30. This restricted us to choose the loop bound, k = 30.

Table 5.13 shows the performance of the NDS-CP algorithm solving this problem.

NDS-CP		
Problem Size N = 149		
Subalgorithm	cycles$*10^3$	fraction
RTS	22,488	17.76
Main	6,892	5.44
Feasible	962	0.76
C-C-SPC-F	3,983	3.15
ZST-HM	31,841	25.15
cPivots	2,240	1.77
Second Phase	7,541	5.96
Idles	50,660	40.01
Total	126,607	100.00
Utilization	68,406	54.03
Critical Path	126,607	
Exec. Time	12.6 sec	

Table 5.13: Breakdown of cycles in NDS-CP on 1pe1is solving 149 nodes graph.

The 1pe1is configuration took 12.6 seconds and most of its time (26% of total cycles) is spent in building the tree (ZST-HM). There are enormous amounts of *idles* (40% of total cycles) and the overall processor utilization is only 55% of total cycles. The reason is that most of the phases of the hierarchical merging merge only a few pairs concurrently. Hence, the overhead of spawning useless processes is quite high. Moreover, it interacts with RTS heavily which requires a significant amount

of parallelism.

Table 5.14 shows the performance of NDS-SP algorithm on Monsoon solving the same problem.

NDS-SP		
Problem Size N = 149		
Subalgorithm	cycles$*10^3$	fraction
RTS	24,907	17.28
Main	7,011	4.86
Feasible	961	0.67
C-C-SPC-F	3,997	2.77
ZST-HM	31,851	22.09
sPivots	14,452	10.02
Second Phase	8,458	5.87
Idles	52,539	36.44
Total	144,176	100.00
Utilization	83,179	57.69
Critical Path	144,176	
Exec. Time	14.4 sec	

Table 5.14: Breakdown of cycles in NDS-SP on 1pelis solving 149 nodes graph

Monsoon(1pelis) took 14.4 seconds and most of its time (22% of total cycles) is spent in building the tree. The 1pelis is *idle* for 37% and the average utilization is only 58%. The major contribution to the *idles* is due to the *ZST-HM* and *sPiovts* due to lack of the work.

Observations

- The NDS-CP performs better than the NDS-SP algorithm because it solves the graph faster than the NDS-SP.
- For 149 nodes graph, the *cPivots* traverses the 0-token spanning tree and merges all the nodes in a cluster without any firing in its first phase. Thus, the NDS-CP executes only one iteration of the optimization phase. In other

words, the initial *bfs* is the optimal solution. Note that the cycles executed by each subalgorithm are almost same except the pivoting subalgorithms. The reason is that the *cPviots* traverses the tree concurrently. However, the *sPivots* does so one zero-edge at a time. Hence, the NDS-CP is faster than the NDS-SP.

- The *ZST-HM* executes more or less—the utilization is about 90% during some phases of its hierarchical merging—sequentially for this graph. Moreover, it interacts with RTS heavily which needs more parallelism. As a result, it contributes a large amounts of *idles*.

5.6.2 Parallelism in Feasible

The ILCWB problem (e.g. 149 nodes DTP) makes use of both portions, initialization and testing, of the Feasible subalgorithm to solve the compaction problem. In this section, we study the parallelism in this step of NDS algorithms.

The testing portion of this subalgorithm waits until the problem is initialized. During the initialization step, each node initializes itself to its most negative edge weight by traversing the list of outgoing nodes independently. During each phase of the testing step, all N nodes work in synchrony, i.e., all nodes may interfere with each other updating its position, however, they synchronize using implicit m-locks.

The initialization portion shows the average parallelism of 21 (see profile 5.13) which reduces to 17 after the testing portion of this subalgorithm while solving this graph. Each peak corresponds to a phase of the testing portion after about 15,000 cycles. The deep valleys between each phase show that each phase is executed sequentially. However, there is significant amount of parallelism during each phase.

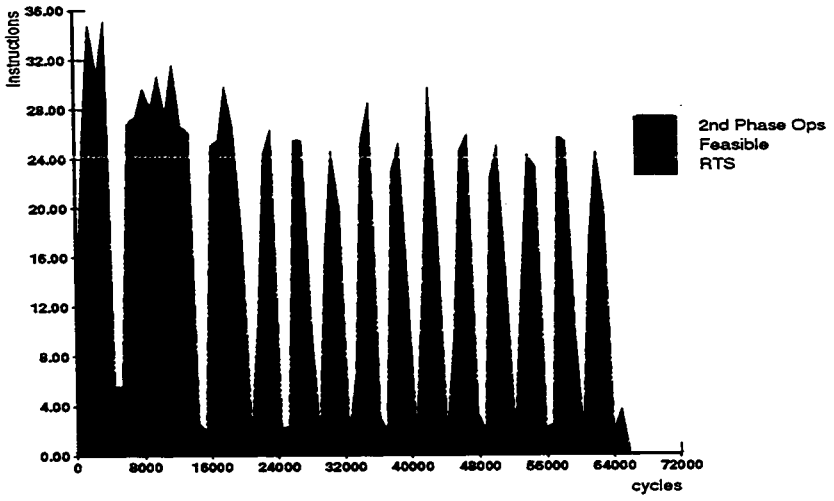

Figure 5.13: Parallelism in Feasible compacting 149 nodes ILCWB problem

5.7 Summary

In this chapter, we evaluated the NDS-SP and NDS-CP by studying their performance on MINT and Monsoon hardware. Our parallelism study reveals that these algorithms interact with RTS heavily and the *finding an initial bfs* portion of then consists a good amount of parallelism to hide latency. However, the hierarchical merging stage of tree-building and both pivoting strategies do not have the parallelism required to hide the latency, hence, to keep the processor pipeline busy. This study also finds that sequential pivoting strategy performs better than the concurrent pivoting strategy for the graphs of size less than 100 nodes due to the poor performance of tree-building approach which is the most time-consuming step in these algorithms.

Chapter 6

Conclusions and Future Work

6.1 Summary

In this thesis, we studied the parallelism in the network dual simplex (NDS) algorithms and also the contributions of Id language's features in revealing the parallelism. To do so, we implemented the NDS algorithm with concurrent pivoting (NDS-CP) using Id and then we used this algorithm to solve the layout compaction & wire-balancing (LCWB) and the integrated LCWB problems formulated as a DTP. The optimization step in the NDS-CP is a repetitive step and involves finding a basic feasible solution (*bfs*), and then performing concurrent pivot steps. The concurrent pivot steps during the concurrent pivoting destroy the basisity of the solution. So, to maintain a *bfs*, NDS-CP repeats the shortest path computations before it starts the next pivoting phase. For our comparative performance study, we implemented this algorithm with a sequential pivoting strategy (NDS-SP). The optimization step in this algorithm differs from the NDS-CP algorithm. In this case, it does not need to perform shortest path computations after each pivoting phase because this strategy maintains the basisity of the solution after each pivot step.

6.2 Review of Results

MINT shows the parallelism of 25 in the NDS-SP; and 20 in the NDS-CP. Their profiles also show that these algorithms interact with RTS heavily. Because RTS operations require a significant amount of parallelism to hide the latency on real hardware, a Monsoon multiprocessor shows a huge amount of idles on all of its configurations. As a result, the processors are not utilized to their capacity.

Through our analysis of the results for a 50 nodes and 500 edges graph, we observed the following facts:

- The overhead due to building the intermediate zero-token spanning tree is very large even though we modified the original method [Cha94]. We believe that it is because of the nature of the problem of building the tree which is inherently sequential.
- There is not enough parallelism in performing concurrent pivot operations.

As a result of these observations, the NDS-SP solves the graph in less amount of time than the NDS-CP for the graphs of size less than 100 nodes. Moreover, these algorithms do not utilize the Monsoon processors efficiently due to lack of the parallelism primarily in the tree-building and pivoting steps.

6.3 Comments

In our development of the solution, we did not pay any explicit attention to parallelism at all, concentrating instead on correctness and clarity of expression. We believed that the implicit way of expressing parallelism due to Id's operational semantics and Id's declarative nature would exploit a huge amount of parallelism from our implementation on Monsoon. However, our dreams were shattered when we looked at the behavior of the NDS algorithm solving the 50 nodes graph on MINT. This blame goes to one or all of the following.

- A naive Id programmer.

  We used all the language features (Higher-order functions, non-strictness, list and array comprehension, etc.) which would play a major role to gain the parallelism in our codes. We used M-structures extensively to avoid threading and extensive copying. However, their use forced us to use explicit barriers in some parts of our code which ultimately serialized our code.

- The algorithm itself.

  Through our parallelism study, we found that tree-building and concurrent pivoting do not offer much parallelism even when they are implemented with a dataflow-style of execution in mind.

- the Id compiler.

  Id's explicit annotations to control the parallelism in an Id program greatly affects its performance because it is very difficult for a programmer to find an optimum value for the loop bounds. A way to automate this process is being studied by Id's compiler researchers at M.I.T.

- The Monsoon architecture.

  The data-structures are stored in the I-structure memory of Monsoon. However, the frame memory per processor can also be used as an I-structure memory. Through our performance study, we found that the number of frame fetch/store (move) operations are very large as compared to the number of I-structure fetch/store operations. The frame fetch/store operations do not cause idles, however, each fetch from a I-structure unit causes an idle because they are satisfied by going to an I-structure unit external to the PE. But the number of idles due to I-structure fetch/store operations are not significant. Thus, we strongly believe that the major contribution to the total amount of idles are due to empty token queues which, in turn, are due to the lack of work

caused by the sequentializing effect of multi-cycle RTS operations. The small frame memory per processor and its hardware hazard are the architectural problems which limit the performance of an Id program. The huge amount of *bubbles* can not be avoided because they occur from the tagged-token dataflow-style of execution adhered to Monsoon.

Finally, the implicit style of parallel programming using functional languages makes it easy to program Monsoon. However, the debugging of an Id program is a nightmare due to lack of I/O facilities (e.g., printing of integer/string data during its execution). The implicit atomic data structure (M-structure) allows the parallel processes to interact more freely, and because their use in a program avoids excess copying, parallel programs can be more efficient.

6.4 Suggestions for further work

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

- Our implementation has not used M-structures of Id to its capacity. Their use forced us to use explicit barriers to synchronize parallel tasks originating from some consecutive for-loop expressions to produce determinate answers which results in excessive serialization in our code. We think that they can be re-written to eliminate these barriers such that these processes may interact properly. However, it may result in excessive congestion on m-locks if M-structures are used. If I-structure are used to exploit producer-consumer parallelism then it may result in excessive copying of data-structures and threading among processes which may put a restriction on the problem size.

- The building of the 0-token spanning tree is an expensive operation due to its sequential nature. It is more critical in the case of the NDS-CP algorithm

because it is executed repeatedly. A better approach of build the tree can be investigated.

Bibliography

[1] Arvind and D. E. Culler. Dataflow architectures. *Annual review of computer science, Vol 1, Palo Alto, CA : Annual Reviews*, pages 225–253, 1986.

[2] P. S. Barth. *Atomic Data Structures for Parallel Computing.* PhD thesis, Massachusetts Institute of Technology, March 1992.

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

[4] Raghu P. Chalasani. *Parallel Netwok Optimization On a Shared Memory Multiprocessor And Applications In VLSI Layout Compaction and Wire-balancing.* PhD thesis, Concordia University, March 1994.

[5] J. Hicks, D. Chiou, B.S. Ang, and Arvind. Performance studies of the monsoon dataflow processor. *Journal of Parallel and Distributed Computing*, July 1993.

[6] R. Paul Johnson. Id world reference manual. Technical report, Massachusetts Institute of Technology, LCS Cambridge MA., September 1990.

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

[8] Motorola , Inc. *Id World User's Manual*, id world 105 sun-4 version edition, 1992.

[9] R. S. Nikhil. Id language reference manual. Technical report, Massachusetts Institute of Technology LCS, Cambridge, MA, September 1990.

[10] R.S. Nikhil. The parallel programming language id and its compilation for parallel machines. *Intl. J. of High Speed Computing*, 5(2):171–223, October 1993.

[11] G. M. Papadopoulos. Program development and performance monitoring on the monsoon dataflow multiprocessor. *In Proceedings of the Workshop on Instrumentation for Future Parallel Computing Systems.*, 1989.

[12] G. M. Papadopoulos and D. E. Culler. Monsoon:an explicit token store architecture. *Proc. 17th Annual Intl. Symposium on Computer Architecture*, pages Pages 82–91, 1990.

[13] S. Sur and W. Bohm. Functional, i-structure and m-structure implementation of nas benchmark ft. In M. Cosnard, G. R. Gao, and G. M. Silberman, editors, *Parallel Architectures And Compilation Techniques, IFIP Transactions.* North-Holland, August 1994.

[14] B. K. Szymanski. *Parallel Functional Languages and Compilers.* ADDISON-WESLEY PUBLISHING COMPANY, ACM Press N.Y., 1991.

[15] K.Y. Tham. Parallel processing for cad applications. *IEEE Design and Test*, October 1987.

[16] Kenneth R. Traub and G. M. Papadopoulos. Overview of the monsoon project. *IEEE*, 1991.

[17] Chvatal Vasek. *Linear Programming.* W. H. Freeman and Company, 1983.

[18] T. Yoshimura. A graph theoretical compaction algorithm. *IEEE International Symposium on Circuits and Systems proceedings.*, 1985.