A Parallel Execution Model for Logic Programming

Thesis by

Peyyun Peggy Li

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technelogy

Pasadena, California 91125

1986

(Submitted April 24, 1986)

Acknowledgment

I would like to thank Alain Martin, my advisor, for his guidance, his constant support and his insights. He guided me through the process of establishing the research ideas, conducting this research, writing the thesis, and polishing it over and over again till its final form. He has not only helped me in this thesis work but also taught me how to organize and present my thoughts clearly. I also owe a special thank to Jan van de Snepscheut for his idea of the Sneptree and for his extensive comments on my thesis.

I would like to thank Lennart Johnsson for his encouragement and his friendship. Being a teacher and a good friend, he led me to the area of parallel processing and taught me how to become a good scientist. I also wish to thank Carver Mead for his warmth and his care. His inspiration always enlightens me. The experience of working with him has been exciting and rewarding.

Thanks to my fellow students, Pieter Hazewindus, Kevin Van Horn, and Steve Burns for their helpful discussions and careful proofreading of the earlier drafts of this thesis. Thanks to Calvin Jackson for his help in solving typesetting problems. He is so patient to answer any questions, trivial or difficult. Also thanks to Arlene DesJardins for her valuable assistance in resolving computing facility problems.

Most of all, I wish to thank my mother and my parents-in-law for their support and love, my husband who shares sorrow and happiness with me, and my son who gives me many sleepless nights and innumerable joyful moments. Without them, this thesis work is meaningless to me.

This research was sponsored by the Defense Advanced Research Projects Agency, ARPA Order Number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Abstract

The Sync Model, a parallel execution method for logic programming, is proposed. The Sync Model is a multiple-solution data-driven model that realizes AND-parallelism and OR-parallelism in a logic program assuming a message-passing multiprocessor system. AND parallelism is implemented by constructing a dynamic data flow graph of the literals in the clause body with an ordering algorithm. OR parallelism is achieved by adding special Synchronization signals to the stream of partial solutions and synchronizing the multiple streams with a merge algorithm.

The Sync Model is proved to be sound and complete. Soundness means it only generates correct solutions and completeness means it generates all the correct solutions. The soundness and completeness of the Sync Model are implied by the correctness of the merge algorithm.

A new class of interconnection networks, the Sneptree, is also presented. The Sneptree is an augmented complete binary tree which can simulate an unbounded complete binary tree optimally. Amongst different connection patterns of the Sneptree, some are regular and extensible so as to be well suited for VLSI implementation. A recursive method is presented to generate the H-structure layout of one type of the Sneptree, called the Cyclic Sneptree. A message routing algorithm between any two leaf nodes of the Cyclic Sneptree is also presented. The routing algorithm, which is of O(n) complexity, gives a good approximation to the shortest path.

The Sneptree is an ideal architecture for the Sync model, in which a dynamic process tree is constructed. With a simple mapping algorithm, the Sync Model can be mapped onto the Sneptree with highly-balanced load and low overhead.

Table of Contents

Acknowledgments iii

Abstract iv

List of Figures viii

Chapter 1 Introduction 1

1.1 Parallelism in Logic Programming 2

1.2 Ensemble Machine and Interconnection Network 4

1.3 A Parallel Execution Model of Logic Programming on a Sneptree 5

1.4 Organization of the Dissertation 7

Chapter 2 The Language and Its Semantics 9

2.1 Introduction 9

2.2 Logic Program and Its Computation Model 10

2.3 Sequential Prolog 15

2.4 CLP and Its Semantics 16

2.5 An Example 19

2.6 Comparison With Other Languages 22

2.6.1 Relational Language 22

2.6.2 Concurrent Prolog 24

2.6.3 Epilog 25

2.7 Summary 26

Chapter 3 The Sync Model 28

3.1 Introduction 28

3.2 Multiple Paths in the Data flow graph 29

3.3 The AND Process and the OR Process 31

3.4 Summary 37

Chapter 4 The Ordering Algorithm 42
4.1 Introduction 42
4.2 Ordering Algorithm 43
4.3 Examples 49
4.4 The Complexity of the Ordering Algorithm 56
4.5 Summary 60

Chapter 5 The Merge Algorithm 62
5.1 Introduction 62
5.2 Base-case Algorithm 63
5.3 General Algorithm 64
5.4 Correctness Proof 69
5.5 Summary 78

Chapter 6 Improvement of The Sync Model 80
6.1 Partially Instantiated Terms 80
6.1.1 Dynamic Links 80
6.1.2 Multiple Paths and Dynamic Links 83
6.1.3 The Extended Sync Model 84
6.2 Stream Parallelism 86
6.3 Tail Recursion Optimization 92
6.3.1 Tail Recursion Optimization with Parallelism 93
6.3.2 Tail Recursion Optimization without Parallelism 96
6.4 Nondeterministic and Deterministic Modes 98
6.5 Summary 99

Chapter 7 The Sneptree 101
7.1 Introduction 101
7.2 Definition of the Sneptree 102
7.3 Mapping of a Binary Tree onto a Sneptree 104
7.4 Layout of a Cyclic Sneptree onto an H-structure Plane 108
7.4.1 Recursive Generation of H-structure Layout 108
7.4.2 Extension of the Sneptree 116
7.5 Summary 117
7.5.1 Comparison with other related works 117
7.5.2 Different Connection Patterns 118

Chapter 8 Routing In A Cyclic Sneptree 121

8.1 Motivation 121
8.2 The Routing Algorithm 123
8.3 Performance 134

Chapter 9 Mapping Of the Sync Model onto the Sneptree 137
9.1 Mapping Criteria 137
9.2 Mapping Algorithm 139
9.3 Mapping Performance 142
9.4 Mapping onto Other Networks 149
9.4.1 Mesh and Torus 149
9.4.2 Boolean n-cube 150
9.5 Summary 151

Chapter 10 Conclusion 153

References 156
Appendix A Simulation Results for the Ordering Algorithm 162

Appendix B Simulation Results for the Mapping Performance 175

List of Figures

Figure 2.1. BNF of a Horn Clause 10
Figure 2.2. An Example of Family Relationships Database 13
Figure 2.3. A Search Tree for "$:-$ *grandparent*$(G, aaron)$." Problem 14
Figure 2.4. Another Search tree for "$:-$ *grandparent*$(G, aaron)$." Problem 15
Figure 2.5. The Process Tree of the Program in Figure 2.1 21
Figure 3.1. A Process with Two Inputs 30
Figure 3.2. An Example with Multiple Path 30
Figure 3.3. The Block Diagram of the AND Process 33
Figure 3.4. The Block Diagram of the OR Process 33
Figure 3.5. The Program for AND-PROC 35
Figure 3.6. The Program for OR-PROC 38
Figure 4.1. An Example with True/False Channels 47
Figure 4.2. Two Refinements of Figure 4.1 47
Figure 4.3. Example 1: Quicksort 49
Figure 4.4. The Clause where the Ordering Algorithm Is to Be Applied 50
Table 4.5. Channel Table (CT) after Step (1) 51
Figure 4.6. Data Flow Graph of *quicksort* 51
Figure 4.7. A Database of Students, Courses and Professors 52
Figure 4.8. A Query Clause upon Which the Ordering Algorithm Is to Be Applied . . 53
Table 4.9. Channel Table (CT) after Step 1 53
Figure 4.10. Data Flow Graph of $query(S, P)$ 54
Table 4.11. Channel Table (CT) after Refinement for Literal (12) 54
Table 4.12. The Refined Channel Table (CT) 55
Figure 4.13. The Refined Data Flow Graph 55
Figure 4.14. Two Worst Case Data Flow Graphs for the Firing Process 57
Table 4.15. The Simulation Results of the Ordering Algorithm 58
Figure 4.16. The Worst Case Data Flow Graph for Detection of Multiple Paths 59

Figure 5.1. Base-case Algorithm: Simple Version 64
Figure 5.2. Base-case Algorithm: Modified Version 65
Figure 5.3. Procedure $cp(e, i)$ 66
Figure 5.4. General Algorithm for Two Buffers 68
Figure 5.5. The Transformation of an AND Process with a Single Input 70
Figure 5.6. The Transformation of the Merge Algorithm with n Inputs 71
Figure 5.7. A Data Flow Graph with Multiple Paths 75
Figure 5.8. A Subtree for $g1 : -p_1, p_2, \ldots, p_n$. 78
Figure 6.1. Data Flow Graph for Clause (6.1) 81
Figure 6.2. Data Flow Graph for Clause (6.2) 82
Figure 6.3. Another Example of Dynamic Links 82
Figure 6.4. Extra Sync Generator Caused by Dynamic Links 84
Figure 6.5. Procedure *output*, Which Handles Partially Instantiated Terms 85
Figure 6.6. Program for MERGE_AND 86
Figure 6.7. Quicksort Program 89
Figure 6.8. The Clause after the Ordering Algorithm Has Been Applied 89
Figure 6.9. The Clause with Stream Variables 89
Figure 6.10. The Process Tree with Tail Recursion 92
Figure 6.11. The Process Tree with Tail Recursion Optimization 93
Figure 6.12. The Process Tree with Tail Recursion Optimization 95
Figure 6.13. Matrix Multiplication Program 95
Figure 6.14. An Example of Tail Recursion Optimization without Parallelism 98
Figure 7.1. A Three–level Sneptree 102
Figure 7.2. A Three–level Cyclic Sneptree 103
Figure 7.3. Two Basic Three–level H–Sneptrees 109
Figure 7.4. Construction of A_{k+1}, A_{k+2} and B_{k+2} Using A_k and B_k 110
Figure 7.5. Construction of A_4, A_5 Using A_3 and B_3 110
Figure 7.6. The Areas and the Longest Wires of the 2-level and 3-level H-Sneptrees . 113
Figure 7.7. The Area of an n-level H-structure Sneptree 113
Figure 7.8. To Find the Longest Wires in Terms of L_n or W_n 115
Figure 7.9. Extension of the Two (m-1)-level Sneptrees to an m-level Sneptree 116
Figure 7.10. A de Bruijn Network with 15 Nodes and Degree 4 118
Figure 7.11. Another Cyclic Sneptree 119
Figure 7.12. A three–level Exchange Sneptree 120
Figure 8.1. Four Possible Routes Between x and y 124

Figure 8.2. The Leaf Nodes with Distance 2 or 3 from Node B 125
Figure 8.3. The Shortest Paths xB and xD 125
Figure 8.4. The Program to Compute the Shortest Route 127
Figure 8.4. The Program to Compute the Shortest Route (cont.) 128
Figure 8.5. The Shortest Path Between x_{i-1} and x_i 129
Figure 8.6. The Programs for Routing Messages 131
Figure 8.6. The Programs for Routing Messages (cont.) 133
Figure 8.6. The Programs for Routing Messages (cont.) 134
Figure 8.7. The Average Routing Distances 136
Figure 8.8. The Average Routing Distances of Shifting by 2^k Operations 136
Figure 9.1. Mapping Example 140
Figure 9.2. Smallest Binary Tree Mapping 141
Figure 9.3. (1) 15-node Cyclic Sneptree 143
Figure 9.4. (2) 15-node Linear Cyclic Sneptree 143
Figure 9.5. (3) 15-node De Bruijn Network 144
Figure 9.6. (4) 15-node Exchange Sneptree 144
Figure 9.7. (5) 15-node Self-Looping Sneptree 145
Figure 9.8. (6) 15-node Planar Cyclic Sneptree 145
Figure 9.9. (7) 15-node Another Cyclic Sneptree 146
Table 9.10. Comparison of Seven Different Connection Patterns 146
Figure 9.11. Edge-to-edge Mapping of a 4-level Binary Tree onto a 4×4 Mesh 150
Figure 9.12. One-to-one Mapping of a 4-level Binary Tree onto a 7×3 Mesh 150
Figure 9.13. One-to-one Mapping from a 3-level Binary Tree onto a 3-cube 151
Figure A.1. The Data Flow Graph of Example 3 165
Figure A.2. The Data Flow Graph of Example 4 166
Figure A.3. The Data Flow Graph of Example 5 167
Figure A.4. The Data Flow Graph of Example 6 168
Figure A.5. The Data Flow Graph of Example 7 169
Figure B.1. The Data-Independent Programs: sort(N) and a(N) 176
Figure B.2. The Program Input of the Simulator 177
Figure B.3. The Process Tree for *sort*(4) 178
Figure B.4. The Process Tree for *a*(4) 179
Figure B.5. The Load-Balancing Factor *LF* of *sort*(*N*) with Algorithm V1 180
Figure B.6. The Load-Balancing Factor *LF* of *sort*(*N*) with Algorithm V2 180
Figure B.7. The Load-Balancing Factor *LF* of *sort*(*N*) with Algorithm V4 181

Figure B.8. The Load-Balancing Factor LF of $sort(N)$ with Algorithm V5 181
Figure B.9. The Load-Balancing Factor LF of $a(N)$ with Algorithm V5 182
Figure B.10. The Load-Balancing Factor LF of $sort(N)$ for Differnt Algorithms . . . 182
Figure B.11. The Average Comm. Overhead CO of $sort(N)$ with Algorithm V1 . . . 183
Figure B.12. The Average Comm. Overhead CO of $sort(N)$ with Algorithm V2 . . . 183
Figure B.13. The Average Comm. Overhead CO of $sort(N)$ with Algorithm V4 . . . 184
Figure B.14. The Average Comm. Overhead CO of $sort(N)$ with Algorithm V5 . . . 184
Figure B.15. The Average Comm. Overhead CO of $a(N)$ with Algorithm V5 185
Figure B.16. The Average Comm. Overhead CO of $sort(N)$ for Different Algorithms 185

Chapter 1

Introduction

With the advent of VLSI and automated design tools, computer designers are no longer limited by the cost of processing logic. As a result, the von Neumann paradigm for computer design, in which a single CPU operates on data serially read from and written into memory, is obsolete for computationally demanding tasks. The designer can now attack problems that once were computationally intractable by implementing systems in which thousands or even tens of thousands of processors cooperate to solve a single problem. The design of such multiprocessor computer systems has been motivated by applications that manipulate a huge amount of information or need massive computations, such as artificial intelligence, simulations, image processing, partial differential equations, or other "hard" scientific computations.

Any multiprocessor system for distributed computation must be designed to allow efficient communication between processors, and between memories and processors, lest the advantages of multiprocessing be negated by inefficient communication. As the number of processors grows, the interconnection design becomes more critical as crossbar or centralized bus schemes become impractical. Therefore, the choice of interconnection networks becomes an important issue in designing a multiprocessor system.

Logic programming is a mathematical formalism suitable for expressing problems requiring deductive reasoning in a high-level non-procedural form. Prolog (PROgramming in LOGic) is a simple but powerful programming language developed at the university of Marseille [41] as a practical tool for "logic programming." From a user's point-of-view, one of Prolog's main attractions is ease of programming. Due to its simple semantics and condense syntax, clear, readable and concise programs can be written quickly with few errors. Prolog is especially suited to "symbolic processing" applications such as natural language systems, algebraic manipulation and compiler writing. There is an increasing tendency to substitute Prolog for Lisp in many AI applications, and to integrate Prolog systems and relational data base systems into so-called knowledge base systems. Yet there is another merit of logic programming; i.e., it does not presuppose a von Neumann computer architecture and is therefore inherently well suited to parallel computations.

1.1 Parallelism in Logic Programming

A logic program consists of a set of logic statements which are in the form of so-called "Horn clauses." (e.g., $B_1 \wedge B_2 \wedge \ldots \wedge B_n \rightarrow A$, in which the left-hand side of the implication, the clause body, is a conjunction of literals, and the right-hand side, the clause head, contains a single literal. There may be many clauses with the same clause head, and a literal is also called a goal.) There are two ways to understand the meaning of a logic program: via its declarative semantics and via its procedural semantics. Since a logic program is actually a set of logic statements, the declarative semantics defines what facts can be inferred from these statements, while the procedural semantics tells us how the program is executed. In procedural semantics, a Horn clause is interpreted as a procedure definition, in which the clause head is the procedure name and the clause body is the procedure body.

The interpretation of a logic program is basically the reduction of the search tree of the goal. The sequential Prolog interpreter simply performs a depth-first search of the search tree. Many efforts have been devoted to increasing the efficiency of the Prolog interpreter. The major trend is to add more control in logic programs so that naive backtracking can be replaced by more complex and more efficient search methods. In a concurrent environment, the search tree can be searched totally in parallel. There are two types of parallelism inherent in a logic program. *OR parallelism* is the parallel execution of the clauses whose heads are unifiable with the goal. *AND parallelism* is the parallel execution of several goals of a clause body. Another type of parallelism — *Stream parallelism* [10] — is the pipelining of structured data, in particular, lists between literal or the same body.

As we will see, OR parallelism is much easier to realize than AND parallelism. Among the different models that have been proposed recently for the parallel processing of logic programs, most of them implement only OR parallelism, [7, 8, 16, 17, 19, 28, 29, 51, 57]. A shared-memory multiprocessor system is usually assumed and structure sharing technique is used. Some of the models claim to handle AND parallelism alone [2, 3, 35], or both AND and OR parallelism, such as [4, 11]. Conery's AND/OR process model [11] implements AND parallelism with a data flow model assuming a message passing multiprocessor system. The same idea was adopted by Borgwardt [3] to implement AND parallelism and stream parallelism. Conery's model is the only concrete model so far that handles both AND and OR parallelism. But his model is conceptually complicated and inefficient due to the backward execution and high run-time overhead.

Another approach towards the concurrent execution of a logic program is to modify and extend sequential Prolog into a concurrent programming language. Several concurrent programming languages for logic programming have been proposed, such as Relational Language [9], Concurrent Prolog [45], Epilog [54, 55], and DELTA-PROLOG [38]. The main features of these languages are summarized as follows:

1. Literals have a producer/consumer relation with respect to shared variables (e.g., read-only annotations in Concurrent Prolog);

2. Guards are introduced to sequentialize the execution of two parts of a body, so that a choice can be made between alternative solutions (e.g., commit operator in both Relational language and Concurrent Prolog).

These languages also implement AND parallelism with a data flow model, but they cannot handle nondeterministic programs effectively. Therefore, OR parallelism is not implemented entirely.

AND parallelism is an important source of parallelism for divide-and-conquer type of applications. But based on the above survey, there is no good solution to carry out AND parallelism. Also, none of the proposed models can exploit all the inherent parallelism satisfactorily. Several processing models have been proposed to accomplish one or two types of parallelism. In most previous works, a shared-memory multiprocessor system is assumed to be the target machine. How to map such a processing model onto a fixed interconnection network is rarely investigated.

1.2 Ensemble Machine and Interconnection Network

Due to the development of VLSI technology, it is now possible to construct powerful computers by connecting thousands of small identical processors into a so-called "processor network" or"ensemble machine" [43]. There is no storage or other system resource that is global or shared by individual processors. Each processor has independent control and local memory. Hence, each processor can run its own program independently and asynchronously. Synchronization and communication are done by message passing between neighboring processors. The computation is distributed over the network. Hence, a high degree of concurrency can be achieved.

Many different interconnection networks have been studied, such as the binary tree [1,5,30,27,47], the mesh, the systolic array [25], the boolean n-cube [44], etc. Some machines are dedicated to some special applications [1,30]; some are designed as a general purpose computing engine [5,27,44]. Basically, a computation proceeds based on a computation graph, such as a two-dimensional grid for matrix operations, a binary tree for divide-and-conquer applications, and a shuffle network for FFT. The best machine for a certain kind of application is the machine whose topology is very close to the computation graph of that application, such that the mapping cost from the computation graph to the machine is relatively low. Besides, the machine size is usually fixed while the problem size varies. Therefore, we have to divide the problem evenly into each node of the implementation network. As a consequence, an important question is brought up: how to map a computation graph onto an implementation network, in particular, how to map an oversized problem onto a fixed size network to keep the load of each processor balanced. In [32], a double-twisted torus is proposed to simulate an unbounded mesh. The torus introduces a homogeneous processor network which relieves the boundary problem from a regular mesh so that a bigger mesh can be mapped onto this network automatically and optimally. In this thesis, we present another homogeneous processor network, called the *Sneptree*. The Sneptree is a class of augmented binary trees with identical nodes. Like the torus, the Sneptree relieves the boundary problem of a binary tree so that a complete binary tree of arbitrary size can be mapped onto the Sneptree optimally.

The binary tree has the property that the distance between any two nodes is at most $2\log_2 n$, where n is the size of the tree. Such a network is called "logarithmic network." The Sneptree is an expanded binary tree with more links in each node. Some connection patterns of the Sneptree are regular and symmetric and hence well suited for VLSI implementation. Furthermore, it can simulate an unbounded binary tree so that it is best for divide-and-conquer type applications. Some other augmented binary tree networks have been investigated in the past, such as the X-tree[15], the Hypertree[18], and the De Bruijn Network[42]. These networks also provide extra links to support other connection patterns such as a hypercube [18] or a ring [15], or to shorten the average distance of two nodes [42]. None of them can simulate an unbounded binary tree properly and at the same time have a regular topology suitable for VLSI implementation.

As mentioned above, the Sneptree is a good architecture for the types of applications in which the computation graph is a dynamic tree. The execution of a logic program as in PROLOG is nothing but building and searching of a search tree, which is ideal to apply to the Sneptree architecture.

1.3 A Parallel Execution Model of Logic Programming on a Sneptree

In this thesis, we propose a new concurrent logic programming language, CLP, which is designed for a message-passing multiprocess system. The execution model of CLP, called the Sync Model, is a data-driven model which realizes both AND-parallelism and OR-parallelism. Our Sync Model is distinguished from other models by one or more of the following features:

1. Full OR Parallelism: Backtracking is replaced by parallel execution of all OR branches. Full OR parallelism is realized by parallel unification of all the unifiable clauses, parallel evaluation of all the guard literals, and parallel execution of all the OR branches which succeed in unification. The execution of each OR branch is independent of the others and the solutions can be produced in any order.

2. Full AND Parallelism Without Binding Conflict: AND parallelism is real-

ized by parallel execution of all the goals in a clause body. A dynamic data flow diagram of the goals is constructed in an OR process if the unification succeeds. The links in the data flow diagram are implemented as unidirectional channels between AND processes. Binding conflict caused by shared variables is avoided by allowing exactly one generator for each shared variable. Other AND processes containing shared variables are suspended until the values of the shared variables are available.

3. Data-driven Model: A solution is sent out immediately after it is generated. With this data-driven model, our process tree produces all the solutions without the user's request and more than one data value may be transmitted along a channel.

The Sync Model constructs a dynamic tree of processes where processes are created and terminated dynamically during computation. The target machine architecture for such application is a message-passing multiprocessor system with processors connected into a regular interconnection network. A new interconnection network, the Sneptree, is found to be an ideal structure for our application.

The Sneptree consists of $2^n - 1$ identical nodes and each node has four links. The links are connected to form an augmented complete binary tree where the outgoing links of the leaves are connected back to all the nodes in the network. We prove that a complete binary tree with arbitrary size can be mapped onto a Sneptree optimally. Hence, the Sneptree is particularly well suited for distributed computations with a tree-structured computation graph, such as divide-and-conquer and backtracking. One type of Sneptree, which contains two disjoint spanning cycles and is thus called a *Cyclic Sneptree*, is of particular interest since it can simulate a fully unbalanced tree optimally, such as a left/right skewed tree.

A recursive method is given to generate the H-structure layout of the Cyclic Sneptree. The number of crossings and the length of the longest wires in the H-structure layout are analyzed. A message routing algorithm between any two leaf nodes is presented. The routing algorithm, which is of O(n) complexity, gives a good approximation to the shortest path. The traffic congestion in the nodes at the upper levels is also significantly reduced compared to the binary tree case.

1.4 Organization of the Dissertation

This dissertation contains three major parts: (1). A parallel computation model for logic programming (Chapter 2 to Chapter 6), (2). A new class of interconnection network, Sneptree, and its properties (Chapter 7 and 8), and (3). The mapping of the parallel computation model of logic programming onto the Sneptree (Chapter 9).

Chapter 2 gives a quick introduction to logic programming and its semantics. A new concurrent language based on logic programming, CLP, and its computation model, the Sync Model, are presented. Furthermore, CLP is extensively compared with other concurrent logic programming languages. Chapter 3 describes the Sync Model in detail. We first address, and propose a solution to, the main problem of the AND/OR parallel model with data flow approach; i.e., the synchronization of multiple partial solutions in the data flow graph. Then, we describe the two different types of processes in the Sync Model — the AND process and the OR process. The two major algorithms used in the Sync Model, the ordering algorithm to construct the data flow graph of the literals in a clause body and the merge algorithm to merge the multiple input streams of a process, are discussed in Chapter 4 and Chapter 5 respectively. In Chapter 6, the Sync Model is extended to implement stream parallelism, tail recursion optimization, and execution mode selection.

Chapter 7 introduces a new class of interconnection network, the Sneptree. The performance of mapping an arbitrary-size complete binary tree onto the Sneptree is analyzed and proved to be optimal. One type of Sneptree, called the Cyclic Sneptree, is studied in more detail. An algorithm is presented to lay out a Cyclic Sneptree into an H-structure plan, in which the area, the longest wire in the layout and the number of crossovers are all within satisfactory ranges. Chapter 8 presents a leaf node routing algorithm to route a message from a leaf node of a Cyclic Sneptree to another leaf node. The routing algorithm is of complexity O(n), where n is the size of the Sneptree. The routing algorithm generates a good approximation to the shortest path.

Chapter 9 presents a mapping algorithm to map the process tree of our Sync Model onto a fixed sized Sneptree. Variation of mapping algorithms with different load balancing heuristics are attempted and found to have no significant difference. Different connection patterns of the Sneptree are also attempted. The Exchange Sneptree, which has the best connectivity from the left half of the Sneptree to the right half of the Sneptree, is shown to have the best mapping performance from the simulation result.

Chapter 10 concludes the thesis and proposes future research directions.

Chapter 2

The Language and Its Semantics

2.1 Introduction

By assigning a procedural semantics to predicate calculus, Kowalski [23] has shown that logic can be used as a programming language. A *logic program* is a set of first-order logic formulas which are called Horn Clauses. Prolog (PROgramming in LOGic) is a language based on logic programming, which was designed and implemented by Colmerauer and Roussel in Marseilles in 1972. Prolog has been applied in a variety of areas such as natural language processing, deductive information retrieval, compiler writing, symbolic algebra and robot problem-solving.

Compared to procedural languages, Prolog is a purely declarative language with simple syntax and semantics. The meaning of a logic program is a formalism capable of representing knowledge and of expressing questions about it. The parallelism inherent in a logic program makes it well suited for a concurrent architecture.

In this chapter, we propose a logic programming language, CLP, and its computation model, called the Sync Model. The two forms of parallelism present in a logic program — AND-parallelism and OR-parallelism — are implemented by the Sync Model assuming a message-passing multiprocessor system. In the remaining sections, we first introduce the syntax and the semantics of a logic program and its computation model. In Section 2.3, sequential Prolog is introduced, emphasizing its distinction from pure logic programming. In

Section 2.4, CLP is described and the Sync Model is presented. In Section 2.5, an example program of family relationships and its step-by-step interpretation in our Sync Model is demonstrated. In Section 2.6, CLP is compared with other concurrent logic programming languages, such as Relational Language, Concurrent Prolog and Epilog.

2.2 Logic Program and Its Computation Model

A *logic program* is a set of first-order logic formulas which are called Horn Clauses. A *Horn Clause* has the form:

$$A :- B_1, B_2, \ldots, B_n. \tag{2.1}$$

The syntax of a Horn Clause can be formally defined in BNF as in Figure 2.1.

```
<clause>::= <head>:- <body>.|<head>.
<head>  ::= <compound-term>
<body>  ::= <literal>{,<literal>}
<literal>::= <compound-term>|¬<compound-term>
<compound-term>::= <functor>|<functor>(<arguments>)
<functor>::= <atom>
<arguments>::= <term>{,<term>}
<term>  ::= <integer>|<variable>|<compound-term>
```

Figure 2.1. BNF of a Horn Clause

A *clause* has two parts: its *head* and its *body*. When its body is empty, the clause is called a *unit clause* and ":- " is omitted. The head is a compound-term and the body is a conjunction of literals, where conjunction is represented by comma. The *term* is the basic data object of the program. A term can be an integer, a variable or a compound term. A *compound term* is a function with or without arguments and arguments are again terms. A function with no arguments is called an *atom*, which is treated as a constant. A nonempty list, $[X|L]$, is a special function which takes two arguments: the first argument X is a term, which is the head of the list; the second argument L is again a list, which is the tail of the list. An empty list is denoted as "[ ]". A literal is a compound term or the negation of a compound term. A literal is also called a *predicate*. To distinguish a variable from an atom, a variable's name is started with a capital letter and an atom is started with a lower-case letter.

There are two kinds of semantics of a logic program, namely, *declarative* and *procedural*. Declaratively, a Horn Clause as in (2.1) can be read as "A is true if B_1 and B_2 and, ..., and

B_n are true." Procedurally, it can be read as "To satisfy goal A, satisfy goals $B_1, B_2, \ldots$, and B_n." A unit clause "A." can be interpreted declaratively as "A is true" and procedurally as "goal A is satisfied." In addition to a set of clauses, a logic program contains a *goal statement* of the form:

$$:-\ A_1, A_2, \ldots, A_n.$$

which is read declaratively as "Are A_1 and A_2, …, and A_n true?" and procedurally as "Satisfy goals A_1, $A_2, \ldots$, and A_n." Each literal A_i in the goal statement is called a *goal.*

A computation is the construction of the proof of a goal statement from the program. It can have two results: success or failure. If the computation succeeds, the values found for the variables in the initial goal constitute the output of the computation. A goal can have several successful computations, each resulting in a different output.

The basic inference rule of logic programming is *unification* and *resolution.* Unification is a generalized pattern matching, which is a method to find the most general unifier (mgu) of two formulas. We shall give the following definitions and then describe the unification algorithm.

A *substitution* is a finite set of ordered pairs $\langle V, t\rangle$, where V is a variable and t is a term which does not contain the variable V. If $\mathcal{F}$ is a formula and θ is a substitution, then $\mathcal{F} \cdot \theta$ is a formula such that any variable V appearing in $\mathcal{F}$ is substituted by $t \cdot \theta$ if $\langle V, t\rangle \in \theta$.

A *substitution pair* μ can be defined on a set of two atomic formulas, $\mathcal{F}$, as follows: (1) If $\mathcal{F} = \{X, Z\}$ is a set of two variables, then $\mu = \langle X, Z\rangle$; (2) If $\mathcal{F} = \{X, t\}$ is a set of a variable X and a term t that does not contain X, then $\mu = \langle X, t\rangle$; (3) If $\mathcal{F} = \{f(t_1, \ldots, t_n\}, f(t'_1, \ldots, t'_n)\}$ or $\mathcal{F} = \{P(t_1, \ldots, t_n\}, P(t'_1, \ldots, t'_n)\}$, then $\mathcal{F}$ has a substitution pair iff at least one of the corresponding terms t_i and t'_i has a substitution pair.

Let $\mathcal{F} = \{P, P'\}$ be a set of formulas and θ be a substitution, Then θ is a *unifier* of $\mathcal{F}$ iff $\mathcal{F} \cdot \theta = \{P \cdot \theta\}$, i.e., $P \cdot \theta = P' \cdot \theta$. Furthermore, $\mathcal{F} = \{P, P'\}$ is a *unifiable set* with a unifier θ, iff there exists a unifier λ, such that $\lambda \cdot \theta = \theta$ for all such θ; λ is called a *most general unifier* (mgu).

The *unification algorithm* for a set $\mathcal{F} = \{P, P'\}$ of formulas is the construction of a sequence of formulas and substitutions $(\mathcal{F}_i, \theta_i)$, such that $\mathcal{F}_{i+1} = \mathcal{F}_i \cdot \mu_i$ and $\theta_{i+1} = \theta_i \cdot \mu_i$, where μ_i is a substitution pair of $\mathcal{F}_i$ and the initial pair $(\mathcal{F}_0, \theta_0) = (\mathcal{F}, \emptyset)$. The sequence $(\mathcal{F}_i, \theta_i)$ will terminate for some finite i. $\mathcal{F}$ is unifiable if $\mathcal{F}_i$ becomes a singleton set and then θ_i is the most general unifier of $\mathcal{F}$. Otherwise $\mathcal{F}$ has no unifier [39].

For instance, let $\mathcal{F} = \{f(X, g(X), X), f(Z, W, a)\}$; we can find a set of substitution pairs:

$$\begin{aligned} \mu_1 &= \langle X, Z \rangle, \\ \mu_2 &= \langle W, g(Z) \rangle, \\ \mu_3 &= \langle Z, a \rangle. \end{aligned}$$

The mgu of $\mathcal{F}$, θ, is derived by $\theta = \mu_1 \cdot \mu_2 \cdot \mu_3 = \{\langle X, a\rangle, \langle W, g(a)\rangle, \langle Z, a\rangle\}$ and $\mathcal{F} \cdot \theta = \{f(a, g(a), a)\}$. Therefore, $\mathcal{F}$ is unifiable with a mgu θ.

The resolution principle for logic programming is as follows: Let G_i be a goal statement "$:-\ A_1, \ldots, A_k, \ldots, A_m$." and C_j be a clause "$A :-\ B_1, \ldots, B_n$."; then G_{i+1} can be derived from G_i and C_j with a mgu θ_i such that

(1). A_k is the selected goal by a selection function,
(2). $A_k \cdot \theta_i = A \cdot \theta_i$,
(3). G_{i+1} is "$:-\ (A_1, \ldots, A_{k-1}, B_1, \ldots, B_n, A_{k+1}, \ldots, A_m) \cdot \theta_i$." and G_{i+1} is called a *resolvent* of G_i and C_j.

A proof of an initial goal G_0 with a program P consisting of Horn clauses $C_1, \ldots, C_n$ is a finite derivation of goals $G_0, G_1, \ldots, G_n$, where each G_{i+1} is a resolvent of G_i and C_j. G_0 is *provable* by P if G_n is reduced to empty.

Nondeterminism is present in a resolution step in two different ways: the selection of A_k in a goal statement G_i, and the selection of C_j among a set of clauses whose heads match A_k.

An Example

The following program is an example of defining family relationships [12]:

The first three clauses are examples of clauses with nonempty bodies. X is the grandparent of Y if X is the parent of someone, Z, who is a parent of Y. The parent relationship

```
grandparent(X,Y):-parent(X,Z),parent(Z,Y).          (1)
parent(X,Y):-mother(X,Y).                           (2)
parent(X,Y):-father(X,Y).                           (3)
father(paul,rob).                                   (4)
mother(mary,rob).                                   (5)
father(rob,bev).                                    (6)
mother(dorothy,bev).                                (7)
father(rob,theresa).                                (8)
mother(dorothy,theresa).                            (9)
father(jeff,aaron).                                 (10)
mother(theresa,aaron).                              (11)
```

Figure 2.2. An Example of Family Relationships Database

is defined in terms of father and mother. The rest of the clauses are unit clauses. We can expand the above database to describe all sorts of family relationships, such as sibling, aunt, uncle, cousin, etc. We can then ask questions to the database. A question "Who is a grandparent of Aaron?" is translated into a goal statement: " :-grandparent(G,aaron)." To answer the above question, we first search the program and find that there is only one clause whose head matches the goal, i.e., clause (1). Applying the unification algorithm, we find the mgu $\theta_1 = \{\langle X, G\rangle, \langle Y, aaron\rangle\}$, and the goal statement is reduced to "$:-\ (parent(X,Z), parent(Z,Y)) \cdot \theta_1$", i.e., G_1:

$$:-\ parent(G,Z), parent(Z, aaron).$$

There are two goals in G_1. If we choose the second goal and the first *parent* clause (clause (2)) in the program for the next resolution step, we find the mgu $\theta_2 = \{\langle X, Z\rangle, \langle Y, aaron\rangle\}$ and G_2:

$$:-\ parent(G,Z), mother(Z, aaron).$$

Again, choose the second goal in G_2. Since there is only one definition of *mother* which matches the goal, i.e., clause (11), we select (11) for the next resolution step. With the mgu $\{\langle Z, theresa\rangle\}$, G_2 is reduced to G_3:

$$:-\ parent(G, theresa).$$

Notice that the second goal in G_2 is reduced to empty and there is only one goal left. Again, we use the first definition of *parent* (clause (2)), G_3 is reduced to G_4:

$$:-\ mother(G, theresa).$$

Finally, with the substitution $\{\langle G, dorothy\rangle\}$, G_4 is reduced to empty. The computation terminates successfully and the answer is found in the substitutions during the computation, i.e., *dorothy*. The original goal statement G_0 and the selection function in each resolution step determine a ***search tree*** containing several possible computations. One path of the tree represents one possible computation and the nodes along the path denote a sequence of goal statements $G_0, G_1, \ldots, G_n$, where G_{i+1} is the resolvent of G_i. Figure 2.3 shows a search tree for the problem just described. The functor's names are abbreviated to save space and the selected goal in each goal statement is underlined. The box, ■ , indicates successful termination and the darkened node, •, indicates that there is no clause whose head matches the selected goal. Any path from the root of the search tree to a box represents a successful computation. Different paths may result in different solutions to the variables of the original goal.

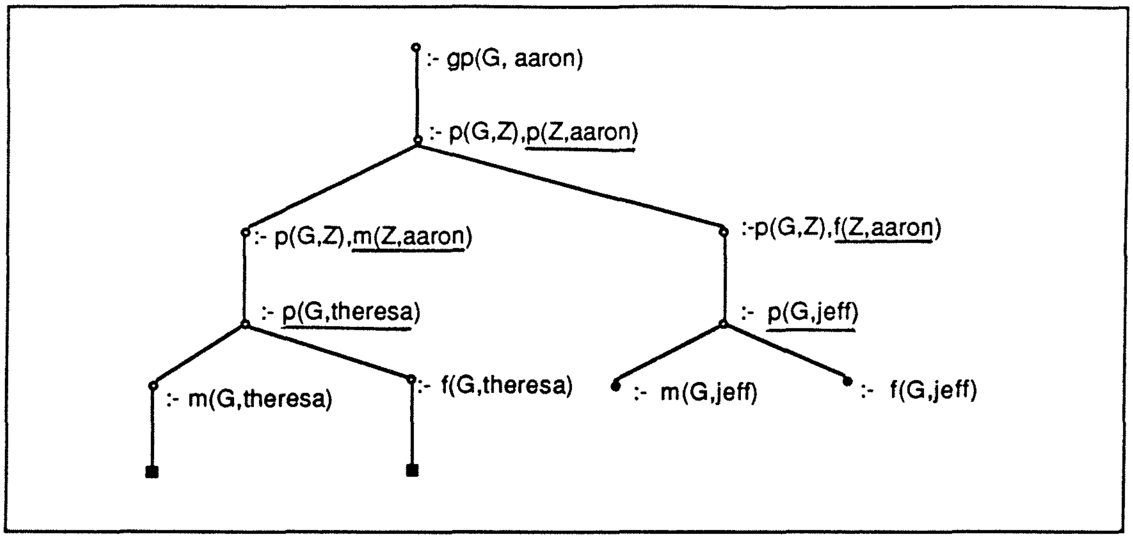

Figure 2.3. A Search Tree for "$:-\ grandparent(G, aaron)$."

With a different selection function, we may generate a different search tree. In the next section, we present another search tree of the grandparent problem. No matter how different two search trees may look, a search tree always contains all the successful computations for an original goal.

In comparison with procedural languages, the collection of clauses with the same functor name in their heads corresponds to the definition of a procedure. A goal is similar to a procedure call. Unification is similar to (but more general than) substitution or binding the actual parameters of a procedure call to its formal parameters.

2.3 Sequential Prolog

Sequential Prolog is a language based on the logic program computation model, especially designed for efficient execution on a von Neumann machine. The order of the goals in a clause body and the order of the clauses in the program determine the control flow of a Sequential Prolog program. Sequential Prolog always selects the leftmost goal in the goal statement for the next resolution. And the selection of the matched clauses is implemented by sequential search and backtracking.

Figure 2.4 is the search tree of the grandparent problem described in the previous section corresponding to Sequential Prolog interpretation. The first goal in a goal statement is always selected for the next reduction. And the search tree is traversed in depth-first manner. The search tree of Figure 2.4 is more complex than the search tree of Figure 2.3. In other words, it takes more time to find an answer.

Additionally, Sequential Prolog provides a "cut" operator ("!") to control the program execution. Cut is inserted in the program as a goal. It is ignored in its declarative semantics. But it affects the backtracking process by inhibiting the backtracking of the goals appearing to the left of the cut and failing the goal which matched the head of the clause containing the cut. Cut is used to speed up the backtracking process by doing away with some fruitless search of the search tree. The semantics of cut is rather obscure.

Sequential Prolog also provides two built-in predicates *assert* and *retract* to manipulate global data of the program. Since a variable within one clause represents one local data object to that clause, the only way to make a data object known to the whole program is by asserting a new unit clause for that data object. Correspondingly, predicate *retract* removes a clause in the program.

In the next section, a new language called CLP is introduced. CLP is another language based on the computation model of logic programming. It is designed especially for parallel execution on a message-passing multiprocessor system.

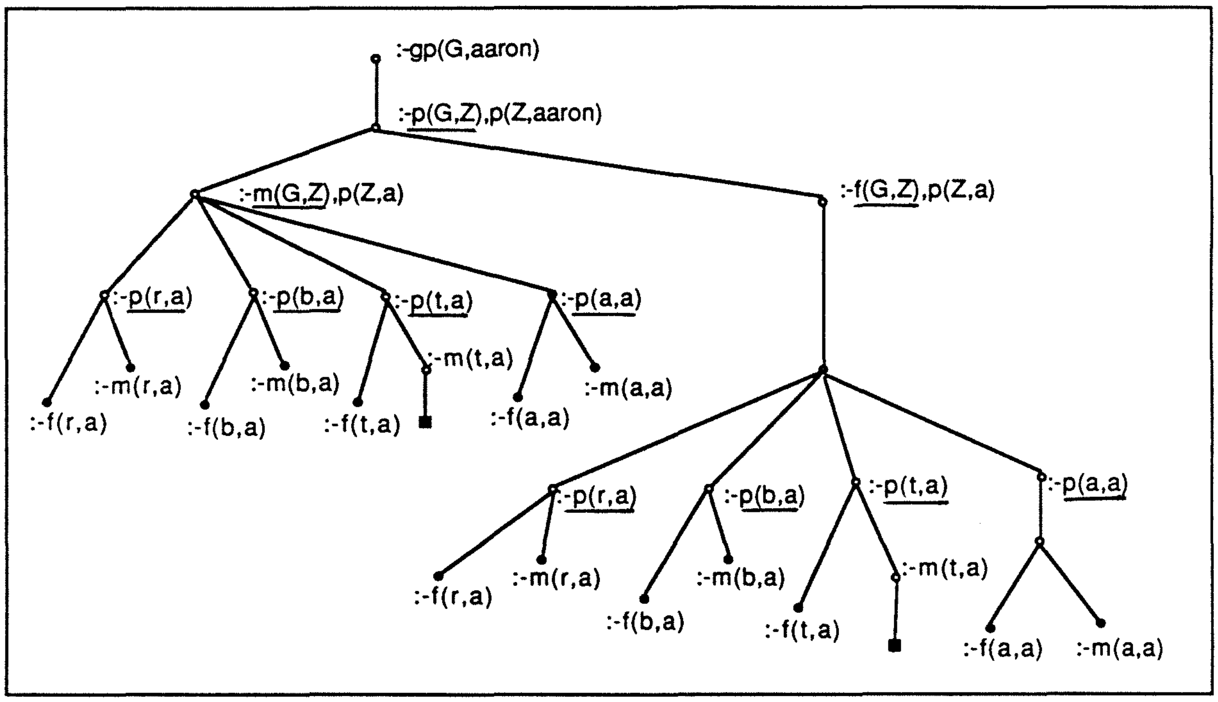

Figure 2.4. Another Search tree for "$:- grandparent(G, aaron)$."

2.4 CLP and Its Semantics

A CLP program is a finite set of *guarded clauses* of the form

$$A :- G_1, G_2, \ldots, G_m \rightarrow B_1, B_2, \ldots, B_n. \tag{2.2}$$

The syntax of a guarded clause can be defined as in Figure 2.1 plus the following definitions:

```
<guarded-clause>::= <clause>|<head>:- <guard>→<body>.
<guard>::= <literal>{,<literal>}
<variable>::= <simple-var>|<simple-var>!|<simple-var>?
```

A guarded clause can be either a Horn clause or a clause with a nonempty guard. The guard of a clause is a set of literals, which are separated from the body by a commit operator, "→." Declaratively, the commit operator reads like a conjunction. The guarded clause in Eq. (2.2) is read as: "A is true, if $G_1, \ldots,$ and G_m, as well as $B_1, \ldots,$ and B_n are true." Operationally, the commit operator forces the sequential execution of the guard and the body. A goal A1 can be reduced to $B_1 \wedge B_2 \ldots \wedge B_n$ if A is unifiable with A1 through a mgu θ and the guard $(G_1, G_2, \ldots, G_m)\theta$ is evaluated to true.

A variable may be either a simple variable, or an *output variable* annotated by a postfix operator "!", or an *input variable* annotated by a postfix operator "?." Variable annotations are not allowed in the clause head. This restriction prohibits annotated variables from appearing in the unification. Therefore, Robinson's unification algorithm [39] can be used directly without any modification. A variable is "shared" when it appears in more than one literal in the body. For a shared variable in the body, at most one literal containing that variable is allowed to have it annotated as output. Such a literal is called the *producer* of that variable, and the literals that contain input variables are called the *consumers* of those variables. The guard may not have any shared variable with the clause head or the body after unification — a guard evaluates to true or false without generating any outputs. But shared variables between guard literals are allowed. Such a syntactic restriction separates the guard and the body into two independent parts, simplifying the implementation of our Model. In each CLP program there is a goal with the form "$:- G$."

Unlike other parallel logic programming languages, the extra language constructs in CLP, the variable annotations and the commit operator, do not affect the semantics of the language. They can be used by the programmer optionally to achieve a more efficient execution under the Sync Model. In order to prevent the semantics from being changed by the commit operator, when the restriction on the variables appearing in the guard is violated, the system simply ignores the commit operator and executes the guard and the body in parallel.

In each step, CLP attempts to resolve all the goals in the goal statement simultaneously and this is called AND-parallelism. In proving a goal, CLP attempts to resolve it with all the unifiable clauses and this is called OR-parallelism. More specifically, CLP realizes AND-parallelism and OR-parallelism inherent in a logic program by breadth-first search of its search tree.

The computation model of CLP, called the Sync Model, is a *process model.* Two types of processes are created and terminated dynamically during the computation. An *AND process* corresponds to a goal, and an *OR process* corresponds to a clause that is used to reduce a specific goal. A tree of interleaved AND and OR processes, called the *process tree*, is constructed corresponding to the AND/OR tree of the program. The initial goal is assigned to an AND process, which becomes the root of the process tree. For each clause whose head is unifiable with the goal of an AND process, one OR process is spawned to carry out the unification and the reduction of this OR clause. After unification succeeds in an OR process, the reduction of the goal is carried out by spawning one AND process for each literal in the body and then reducing the goals in the AND processes concurrently. If the clause in an OR process has a nonempty guard, a set of AND processes is spawned for each goal in the guard first. When all the AND processes for the guard terminate successfully, the OR process may spawn processes for the goals in the body and proceed. When any of the guard literals fails, the OR process fails.

AND parallelism is implemented by dynamically constructing the data flow graph of the literals in the clause body. A *binding conflict* occurs when two or more parallel AND processes bind a common variable to different values. To avoid binding conflicts in the parallel execution of sibling AND processes with shared variables, only one AND process is allowed to be the producer of a shared variable. All the other AND processes that also contain that shared variable are considered the *consumers* of that variable. A consumer process will suspend its computation until the values of its input variables have been received from their producers. A data flow graph of all the literals in the clause body (so-called AND literals), is constructed such that a node represents an AND literal and an edge is directed from the producer of a shared variable to a consumer of that variable. As we shall see, the ordering algorithm will guarantee that the data flow graph is acyclic so as to avoid deadlock. Communication channels are added into the process tree to model the edges of the data flow graph. With the communication channels between sibling AND processes, the process tree is no longer a tree. We prove later that our process tree generates the same results as the corresponding AND/OR tree.

The input and output annotations in CLP are added to the program optionally by the programmer to help constructing the data flow graph such that a more efficient computation can be achieved. For instance, in the family relationships program shown in Figure 2.2, if we change the first clause into

$$grandparent(X,Y) :- parent(X,Z?), parent(Z!,Y).$$

i.e., if we specify the second literal in the body as the producer of shared variable Z and the first one the consumer, then the computation of query "$: -$ *grandparent*$(G, aaron)$." becomes more efficient (Figure 2.3) than the Sequential Prolog interpretation, which always selects the leftmost goal (Figure 2.4). Without explicit variable annotations, the "left to right" order of the AND literals is used for selecting the producer of a variable in CLP. The explicit variable annotation should fulfill the two restrictions on the data flow graph: one producer per variable and acyclicity of the data flow graph. These can easily be checked syntactically.

Parallel execution of different OR processes may produce multiple solutions for the output variables of their father AND process. Those multiple solutions will be transmitted along the communication channels. Hence, we need some mechanism to synchronize the multiple inputs of a given AND process originating from different sources. In our computation model, any process that generates or collects a solution transmits the solution without requiring a request. Hence, our model can be viewed as a *multiple-solution data-driven model.* With this synchronization mechanism, we are able to incorporate both AND parallelism and OR parallelism without any form of backtracking.

In the next section, the family relationships program shown in Section 2.2 is used again to explain the Sync Model. The detail description of AND processes and OR processes can be found in the next chapter. Moreover, two main algorithms, the ordering algorithm in which the data flow graph of the goals is constructed, and the merge algorithm in which multiple input streams of a given process are merged, are described in Chapter 4 and 5, respectively.

2.5 An Example

A process tree for the original goal *grandparent*$(G, aaron)$ is constructed as shown in Figure 2.5. Each OR process is marked by the clause number which is to be unified in that process. Each AND process is marked by the goal passed from the father node when the AND process is invoked. The predicate names are abbreviated to save space. The values along each edge in the process tree are the variable bindings transmitted through that channel. All the values along the vertical edges are going upwards. The values along the horizontal edges flow following the arrows. A shaded node represents a failure node.

An AND process is assigned to the original goal, which becomes the root of the process tree. One OR process (1) is invoked by the root because there is only one clause with head *grandparent* in the definition. The OR process succeeds in unification with substitution $\{\langle X, G\rangle, \langle Y, aaron\rangle\}$ and then invokes two AND processes, p1 and p2, one for each literal in the clause body (AND parallelism). Since there is one shared variable Z between two AND literals, we construct a channel between the two AND processes directed from p2 to p1: p2 is the producer of variable Z; p1 has to wait until it receives the value of Z †. Process p2 then invokes two OR processes (2) and (3) for the two clauses with head name *parent* in the definition. Both OR processes succeed in unification and the OR parallelism is now demonstrated. The first OR process (2) invokes one AND process for goal $mother(Z, aaron)$, so as the the second OR process (3) for goal $father(Z, aaron)$. The AND process containing $mother(Z, aaron)$ invokes four OR processes to unify the goal with clauses (5), (7), (9) and (11) respectively. Only process (11) succeeds in unification with binding $\langle Z, theresa\rangle$. Similarly, the AND process containing $father(Z, aaron)$ invokes four OR processes and only one OR process successfully terminates with binding $\langle Z, jeff\rangle$. The two bindings of Z are sent back to the AND process p2. Upon receiving the bindings of Z, process p2 then sends them to process p1. The AND process p1 first binds Z to *theresa* and expands the process tree similarly to p2. Two values of G are computed by the two OR subtrees of p1, i.e., *dorothy* and *rob*, and returned to the root. After all the processes in the subtree of p1 terminate, p1 binds Z to the second value it receives, i.e., *jeff*, and repeats the computation. This time, all the computations in the subtree of p1 fail and no answers are produced.

A leaf node of the process tree is either an OR process which fails to unify, or an OR process corresponding to a unit clause, or an AND process corresponding to a built-in predicate. An OR process containing a unit clause returns the variable bindings to its father AND process and terminates if it succeeds in unification. An AND process corresponding to a built-in predicate evaluates the predicate directly and sends the variable bindings to proper destination processes. A non-leaf AND process succeeds when at least one of its OR descendants succeeds. It receives the bindings of its output variables from the

† the direction of channel Z is the same whether it is determined by the ordering algorithm or specified by the programmer. The ordering algorithm selects process p2 as the producer because there is one constant argument in the second *parent* predicate.

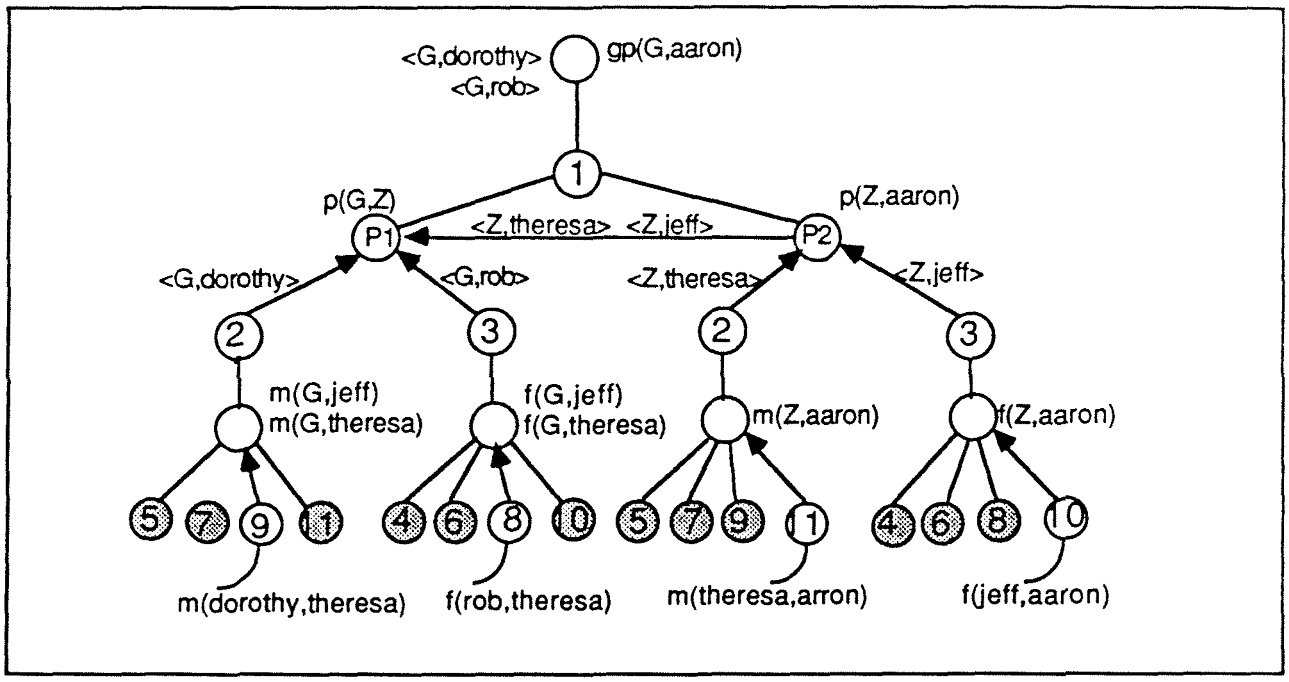

Figure 2.5. The Process Tree of the Program in Figure 2.1

descendants and sends them out to its father and all the sibling consumer processes of its output variables. A non-leaf OR process succeeds when all its descendant AND processes terminate successfully. It merges the results received from its descendants and then sends them to its father.

Without getting into much detail, this sample program shows how our Sync Model resolves a CLP program. We shall identify the following characteristics of our Sync Model:

1. Full OR Parallelism: Backtracking is replaced by parallel execution of all the OR branches. Full OR parallelism is realized by parallel unification of all the unifiable clauses, parallel evaluation of all the guard literals and parallel execution of all the OR branches which succeed in unification. The execution of each OR branch is independent of the others and the speed of each OR branch is immaterial.

2. Data flow model to exploit AND parallelism: AND parallelism is realized

by parallel execution of all the literals in a clause body. A dynamic data flow graph of the goals is constructed in an OR process when the unification succeeds. The links in the data flow graph are implemented as unidirectional channels between AND processes. Binding conflicts caused by shared variables are avoided by allowing exactly one producer for each shared variable. Other AND processes containing these shared variables (consumer processes) are suspended until the values of the shared variables are available.

3. Multiple-solution data-driven model: A solution is sent out immediately after it is generated. All the solutions will be generated by the process tree without the user's request. A user usually doesn't know how many solutions there are for a given nondeterministic program. A demand-driven model always returns one solution and a user can't tell from the result whether or not there are more solutions unless he asks. The data-driven model returns all the solutions to the user; therefore it provides more information than a demand-driven model.

2.6 Comparison With Other Languages

Unlike other parallel logic programming languages, the extra language constructs in CLP, the variable annotations and the commit operator, do not affect the semantics of the language. They may be used by the programmer optionally to achieve a more efficient execution under the Sync Model. The syntax of CLP is very similar to Concurrent Prolog [45] or Relational Language [9], but its execution model is quite different from either of the two. In the following, we compare CLP with other concurrent logic programming languages.

2.6.1 Relational Language

CLP originated from the idea of Relational Language designed by Clark and Gregory [9]. The guarded clause and the variable annotations in CLP are adopted from Relational Language. Besides, both languages restrict themselves to one producer per shared variable and use communication channels to model the data flow between AND literals. But, there are some major differences between the two languages:

1. The guard must be *ground* in Relational language; i.e., there are no variables in the guard when it is invoked. In CLP, we lift the above restriction such that local variables are allowed in the guard literals. But variables that also appear in the body or remain uninstantiated in the clause head are not allowed. Moreover, the guard literals in Relational Language are executed sequentially, while in CLP the guard literals are executed in parallel and the execution of guard literals is the same as the execution of the body.

2. In Relational Language, the body of a guarded clause consists of a number of parallel processes, in which each has a set of goals executed sequentially. Hence, a user has to determine which part of the program should be executed in parallel and which part sequentially. In CLP, each AND process contains only one goal and the conjunction is always interpreted as parallel execution of AND processes. Hence, a user need not be aware of the concurrency of the program, and the system will exploit any inherent concurrency within the program.

3. In Relational Language, the output annotation is mandatory, while in CLP variable annotations are optional. When variable annotations are not provided by the programmer, an ordering algorithm will determine the producer and the consumers of a shared variable. Therefore, a pure logic program (without cut and not assuming the order of the clauses) is a subset of CLP, which can be executed with our Sync Model.

   The above three points are the syntactic difference between Relational Language and CLP. There also exist major differences between their execution models.

4. Relational Language can handle only "don't care" non-determinism, while CLP can handle both "don't care" and "don't know" non-determinism, [24]. When there is more than one candidate clause that may be used to resolve a goal, the first clause that evaluates the guard to true is selected in Relational Language. In our Sync Model, all the candidate clauses will be executed and multiple solutions may be generated. Strictly speaking, Relational Language cannot handle a non-deterministic program properly. It may return a wrong answer when it makes an incorrect choice among several candidate clauses.

5. In Relational Language, the resolution of a goal is equivalent to the reconfiguration of

the process network. During each reconfiguration, new processes and new input/output channels are created and old processes are destroyed. Such an execution model is difficult to map onto a fixed connection architecture. In our Sync model, the resolution of a goal is equivalent to spawning descendant processes. A tree of processes is constructed when the computation proceeds. Such a tree structure is easily mapped onto a fixed connected architecture, such as a Sneptree (see Chapter 9).

6. The computation model of Relation Language does not allow a variable to be bound to a *partially instantiated term*, i.e., a term containing variables. Such a restriction is not practical because partially instantiated variables are very likely to happen in any logic program. The Sync Model handles variables which are partially instantiated by adding dynamic links to the data flow graph of AND literals (see Chapter 6).

2.6.2 Concurrent Prolog

Concurrent Prolog [45] is very similar to Relational Language in both syntax and semantics. Concurrent Prolog also has guarded clauses, commit operator, and read-only annotations. It also creates one process per goal. The reduction of a goal is equivalent to reducing a process by a system of processes. A process is suspended when it attempts to unify a read-only variable to a non-variable term. Concurrent Prolog differs with CLP in the following items.

1. In Concurrent Prolog, a process is suspended when it attempts to unify a read-only variable with a non-variable term. In our model, a process is suspended when waiting for an input from an input channel. In our approach, input variables are bounded before the unification so that the unification rule is not changed. In Concurrent Prolog, the unification rules are modified to handle variable annotations. As a consequence, the variable annotations may be propagated to other non-annotated variables and a read-only variable may get instantiated in a unification.

2. Concurrent Prolog assumes a shared-memory multiprocess system, so that a global binding environment is maintained and referred by all the parallel processes. CLP assumes a message-passing processor network without shared memory. The former architecture cannot eliminate the Von Neumann bottleneck, and the later one with

fixed topology is more suitable for VLSI implementation. Therefore, we prefer the second architecture to the first one.

3. Concurrent Prolog doesn't restrict itself to one producer per shared variable. Any number of parallel processes can be the producers of a shared variable as long as the variable is not read-only annotated in those processes. Binding conflicts may occur and they have to be detected by the system. It involves memory locking and processes racing problems which have to be handled carefully to assure the correct execution of the program. In CLP, the system will select one producer for each shared variable with or without the programmer's annotations. The advantage is that CLP can be treated either as a pure logic program or as a concurrent language, so that different programmers can use it in different ways.

4. Like Relational Language, Concurrent Prolog cannot handle nondeterministic programs.

2.6.3 Epilog

Epilog [54, 55] is again a concurrent logic-programming language which utilizes a data-driven execution strategy and is designed specifically for use on a tightly coupled multiprocessor. The technique of realizing AND parallelism and OR parallelism is similar to our Sync Model. But as the computation model was not explained clearly in the literature, we can only compare the syntax of Epilog with CLP.

Epilog assumes a concurrent computation model of a logic program. Then Epilog is forced with extra syntactic constructs to provide sequential execution control. Fixed sequence constructs (CAND and COR) are used to simulate cut operation and the ordering of the clauses in Sequential Prolog. Data-dependent constructs (threshold functions and variable annotations) are used to establish the data flow model.

Syntactically, CLP is not as general as Epilog. The two languages are designed based on a different philosophy. We intend to provide a logic-programming language with minimal syntactical changes, which is suitable for parallel execution. Epilog is designed as a concurrent language so that a user can specify different control strategies based on the understanding of the problem. Unfortunately, Epilog is no longer a Horn clause logic.

In general, the commit operator in CLP has the same semantics as conditional AND (CAND) in Epilog, but Epilog allows more than one CAND in one clause. Conditional OR (COR) is not supported in CLP. COR is used to simulate the ordering of the clauses in Sequential Prolog. Since nondeterminism is a basic feature for any concurrent language, we decided not to implement COR in CLP. The input/output annotations are available in both languages. Epilog has an extra annotation, "bound to atom," which we found useless and therefore did not implement in CLP.

In conclusion, all four logic languages (CLP, Relational Language, Concurrent Prolog, and Epilog) have somewhat similar syntax, but different computation models. Although the Sync Model is dedicated to CLP, it is not difficult to apply it in implementing the other languages.

2.7 Summary

CLP is a concurrent logic programming language designed for a message-passing multiprocess system. Two syntactic constructs, the commit operator and input/output annotations are added to the language to control the execution of a program. Unlike other parallel logic-programming languages, the extra language constructs in CLP do not affect the semantics of the language. They can be used by the programmer optionally to achieve a more efficient execution under the Sync Model.

The execution model of CLP, called the Sync Model, is a data-driven model which realizes both AND-parallelism and OR-parallelism of a logic program. For a nondeterministic program, all the solutions of the program will be generated by the Sync model without explicit request.

The Sync model contains two kinds of processes, AND processes and OR processes. Each process has its own local memory and independent control. Each process has exactly one father process except for the root process which is connected to the outside world. Each process may have as many descendants as required. The channels between an AND process and an OR process are bidirectional and messages can be transmitted upward or downward along the channels. The AND processes may also have "horizontal" channels to provide communication between sibling AND processes. Those channels are unidirectional. The

AND processes and the OR processes are alternately connected into a dynamic tree that realizes the AND/OR proof tree of a CLP program. An AND process executes one literal in a clause body by evoking OR processes to do the unifications and collects the solutions from all the OR branches. The OR process resolves one OR branch in the AND/OR proof tree by evoking the AND processes for each goal in the clause body if the unification of the clause head with the goal passed by its father AND node succeeds and all the goals in the guard terminate successfully.

Chapter 3

The Sync Model

3.1 Introduction

In this chapter the parallel execution model for CLP, the Sync Model, is described. The Sync Model searches the AND/OR proof tree of a CLP program in breadth-first manner. AND processes and OR processes are interconnected into a dynamic tree. Sibling AND processes are also connected so that the shared variables can be transmitted through the channels. Each process is running independently. Processes communicate and synchronize through message passing.

AND parallelism is achieved in our model by performing an ordering algorithm to construct the data flow graph of the AND literals in a clause body and then mapping each arc in the graph onto a communication channel between AND processes. An AND process that has more than one input channel synchronizes its multiple inputs by performing a merge algorithm on the input streams with special synchronization signals. The ordering algorithm and the merge algorithm are the two major algorithms in the Sync Model.

In a logic program, the producer of a shared variable may bind the variable to a partially instantiated term, i.e., a term containing some other variables. Therefore, the data flow graph of the AND literals is changed dynamically during the computation. The basic Sync Model described in this chapter does not allow the above case because the above situation rarely happens in practice. Such a restriction makes the model simple, clean and easy to understand. But we do propose a solution to this problem in Section 6.1 and present the program for the complete Sync Model which handles partially instantiated terms.

Stream parallelism is not implemented in the basic Sync Model, either. It is an important source of parallelism when two AND processes are pipelined via a shared "list variable." In Chapter 6, the basic Sync Model is further extended to implement stream parallelism and tail recursion optimization.

In Section 3.2, we describe, and propose a solution to, a general problem in the parallel models of logic programming — the synchronization of multiple partial solutions produced by sibling AND processes. In Section 3.3, the functions of OR processes and AND processes are described. In Chapter 4, we describe the ordering algorithm and in Chapter 5, the merge algorithm.

3.2 Multiple Paths in the Data flow graph

The Sync Model generates all the solutions to a nondeterministic program. In other words, several values for a variable may be transmitted through the communication channels in the data flow graph. If one AND process consumes two inputs from two different sources, such as process a in Figure 3.1, the two inputs are combined to form all the input combinations of process a. For instance, if two input values x_1, x_2 are received from one input channel and three input values y_1, y_2, y_3 are received from the other channel, then the legal input combination of process a should be the Cartesian Product of the two input streams, i.e., $(x_1, y_1), (x_1, y_2), (x_1, y_3), (x_2, y_1), (x_2, y_2), (x_2, y_3)$. There is one exception: when the two input streams originate in the same process, the input combination of a is a set of Cartesian Products over certain portions of the two input streams that derive from the same output of the common ancestor. In the sequel, we call a set of paths that have the same starting process and the same ending process a *multiple path* between these two processes. In Figure 3.2, there are two paths (a, b, d) and (a, c, d) between process a and process d. If process a binds (X,Y) to (x_1, y_1) and (x_2, y_2), process b binds T to t_1 and t_2 with input x_1, t_3 with input x_2, and process c binds S to s_1 and s_2 with input y_1, s_3 and s_4 with input y_2, then the legal input combination for process d should be $(t_1, s_1), (t_1, s_2), (t_2, s_1), (t_2, s_2), (t_3, s_3), (t_3, s_4)$ instead of the full Cartesian Product of the two input streams. Observe that the first four input pairs of process d are derived from the input (x_1, y_1) from process a and the remaining two input pairs are derived from (x_2, y_2). Because the two inputs of process d originate in the same process a, we shall form the Cartesian Product over the portions of the input streams that are generated by the same output pair of process a, e.g., (t_1, t_2) and (s_1, s_2), or (t_3) and (s_3, s_4). In order to derive the correct input combination, we mark process a as a Sync generator and separate each output of process a with a special Sync signal. The Sync signals are then propagated through processes b and c and reach process d in both inputs. Finally, process d detects the same Sync signals in both inputs and then forms the Cartesian Product over the input portions that are enclosed by the corresponding pair of Sync signals.

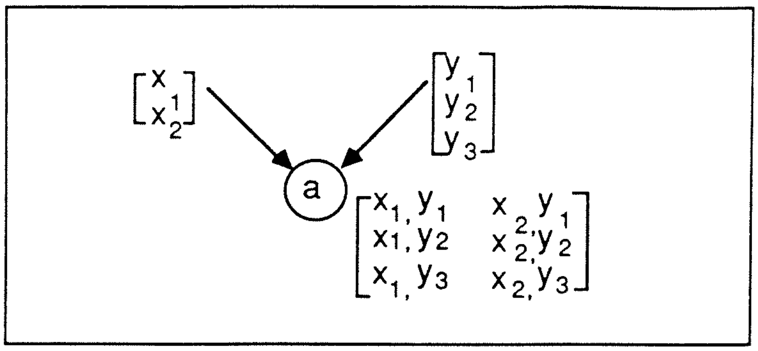

Figure 3.1. A Process with Two Inputs

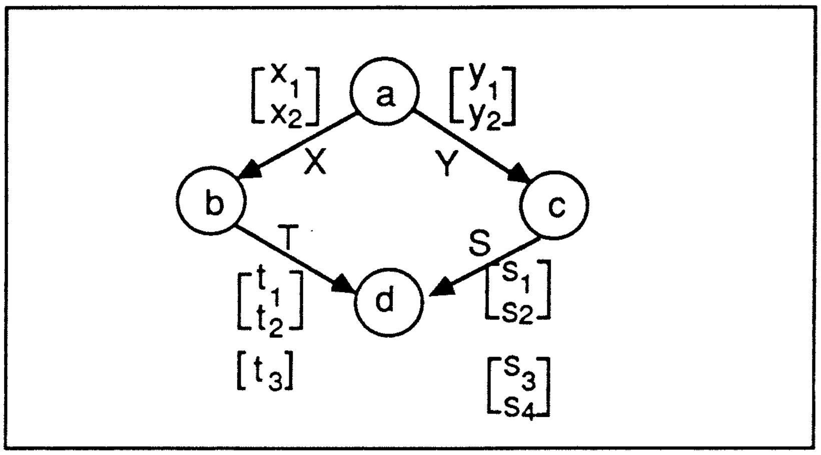

Figure 3.2. An Example with Multiple Path

After the data flow graph has been constructed, we determine all the multiple paths in the graph and mark the starting nodes of those paths as Sync generators. Different Sync generators generate different Sync signals. A process that receives two or more inputs from different channels merges the input streams according to the Sync signals carried in each input stream. The Sync signals may be duplicated during the merge process when they are nested in other Syncs. In the above example, process a is a Sync generator; hence, the output streams generated by process a should be $(S_a, x_1, S_a, x_2, \text{END})$ and $(S_a, y_1, S_a, y_2, \text{END})$, respectively, where S_a represents a Sync signal generated by process a and "END" represents a special signal indicating the end of the stream. Likewise, the output streams of process b and process c should be $(S_a, t_1, t_2, S_a, t_3, \text{END})$ and $(S_a, s_1, s_2, S_a, s_3, s_4, \text{END})$, respectively. Therefore, the input combination of process d becomes $(S_a, (t_1, s_1), (t_1, s_2), (t_2, s_1), (t_2, s_2), S_a, (t_3, s_3), (t_3, s_4), \text{END})$. Once a Sync signal is generated, it is propagated to (and may be duplicated in) sibling AND processes through the communication channels in the data flow graph. The Sync signals will be removed at the father OR process before the output streams are sent out to higher level AND processes. Therefore, the Sync signals are local to the OR process and its AND descendants.

3.3 The AND Process and the OR Process

We now describe the functions of the AND processes and the OR processes. Let AND_PROC and OR_PROC be the two routines running in the AND process and the OR process, respectively. Besides AND_PROC, there is another routine MERGE_AND in an AND process. MERGE_AND is created by the AND_PROC when an AND process contains at least one input variable and therefore a merge operation is needed to construct the input queue. MERGE_AND puts the input combinations into a queue and AND_PROC takes the entries out of the queue. The two routines run concurrently so that AND_PROC can be started as soon as the first input combination is produced by MERGE_AND. Similarly, there is a routine MERGE_OR which works as a coroutine to OR_PROC, in each OR process. MERGE_OR is created by OR_PROC when the OR process is ready to receive answers from its descendants. OR_PROC receives the answers from its descendants and puts the answers into input buffers; MERGE_OR merges the answers from the input buffers and sends the result to the father process. Again, the two routines are running concurrently. Both MERGE_AND and MERGE_OR merge the input bindings in the input buffers. As we shall see in Chapter 4, the two merge algorithms are slightly different.

There are four different types of messages transmitted between processes. Messages *kill* and *spawn* are sent downwards, and messages *fail* and *binding* are sent upwards along vertical channels (i.e., channels between an OR process and an AND process). Only one type of messages, *binding*, is transmitted along horizontal channels between sibling AND processes.

Kill is an interrupt message. It is generated by an OR process that has received a *fail* message from one of its descendants. This message is propagated to all the descendants of the OR process and kills all the active descendant processes.

Spawn is an invoking message. It invokes a new descendant process for the current process. This message is associated with the information of process type (AND process or OR process), process id, the goal to be executed, etc.. Once a process is spawned, it starts running until it terminates normally or it is killed.

Fail is an answer message. Whenever a process fails (either by the failure of unification or the failure of its descendants), it sends a *fail* message to its father.

Binding is another answer message. The message carries either a Sync signal, an "END" signal or a set of variable bindings in the form of $\langle X, t \rangle$. It is generated by a process when the process resolves its goal successfully.

Messages *kill* and *spawn* are used to control the creation and termination of processes. Messages *fail* and *binding* are the data messages sending the results up.

The horizontal binding messages are sent out by AND_PROC and received by MERGE_AND. The vertical messages are sent and received by either AND_PROC or OR_PROC, except that the *binding* message transmitted upwards from an OR process is sent by MERGE_OR.

The structures of the AND process and the OR process, including the four routines and their input and output channels, are illustrated in Figure 3.3 and Figure 3.4 , respectively.

Figure 3.5 shows the program for AND_PROC. The arguments after the routine name are the arguments passed by its invoker when the process is first invoked. The AND_PROC has four arguments. The first argument is the goal to be executed. The second argument is a Sync attribute indicating whether or not this process is a Sync generator. The third argument is the identifier of its goal, which is determined by the ordering algorithm, and

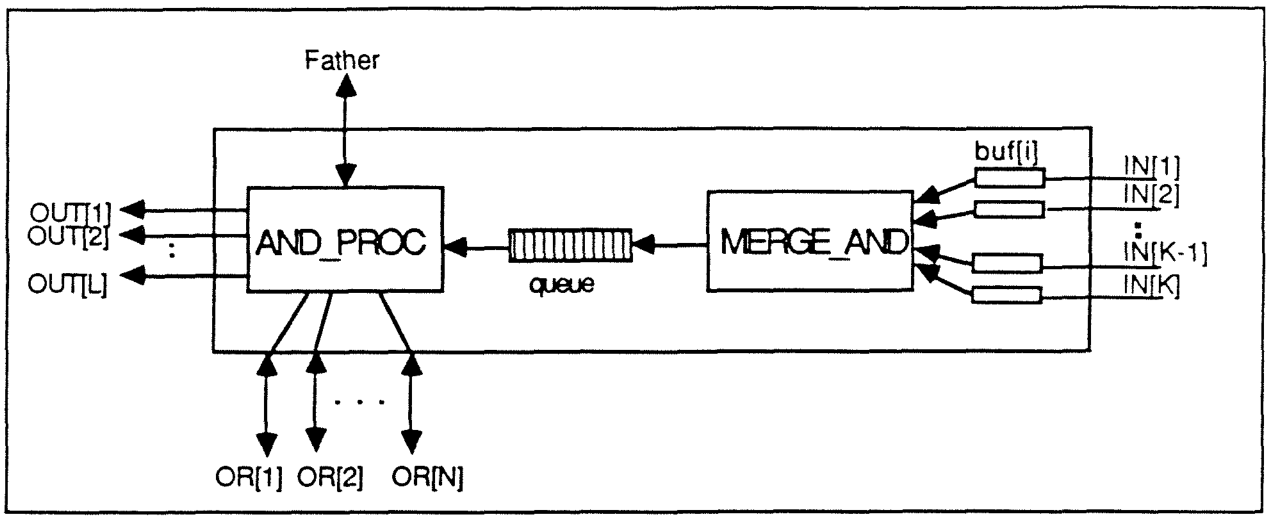

Figure 3.3. The Block Diagram of the AND Process

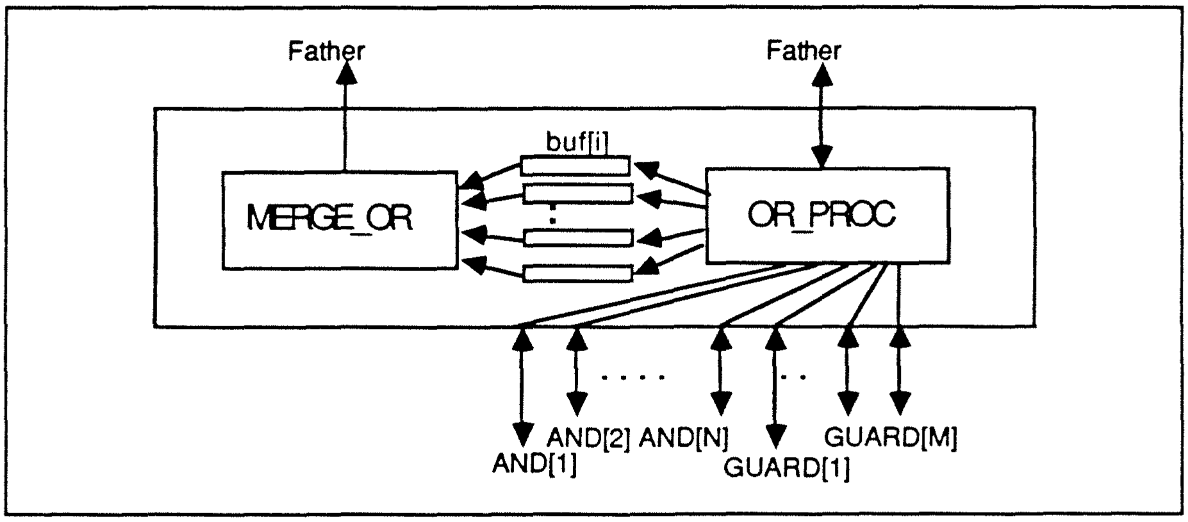

Figure 3.4. The Block Diagram of the OR Process

implies the partial ordering of the literals in the data flow graph. And the last argument is a list of (variable, channels) pairs, which is the channel information for all the output variables in the goal. A *kill* message received from the Father process interrupts all the ongoing processing. The AND process propagates the *kill* message to all its active descendants, and terminates. During its normal processing, AND_PROC first checks to see if there is any input variable in its goal ("if_input_vars_in_goal"). If there exists any input variables, it creates MERGE_AND in the same AND process and passes the list of input variables to MERGE_AND ("start"). MERGE_AND merges the inputs from the input channels and puts the results into a queue; AND_PROC then takes the inputs out of the queue one at a time ("dequeue"). If input is an "END" signal, the queue is empty and AND_PROC propagates the "END" signal through all its output channels † ("send_to_all_channels"), and terminates. If the input is a Sync signal, the routine propagates the Sync signal and gets the next input in the queue. Otherwise, the input is a set of variable bindings containing all the input variables in the goal.

Now, the input values are substituted for the input variables in the goal ("bind_input_to _goal") to form a new goal without any uninstantiated input variables. The process then checks to see if the current goal is a built-in predicate (i.e., a system-defined predicate). If so, the predicate is evaluated; the bindings for the output variables are generated and sent to the corresponding output channels. Otherwise, a process list ("OR_list") of length N is created, where N is the number of clauses whose heads match the goal, and N OR processes are spawned with *spawn* messages. These N descendants are named OR[1], OR[2], ..., OR[N].

After all the OR processes have been created, the current process is ready to gather the results from its descendants. The OR_list contains all the active OR descendants. Whenever a descendant OR process fails or terminates, it is removed from the OR_list ("remove"). When the OR_list becomes empty, the gathering step is finished and AND_PROC is ready to handle the next input in the queue.

While the OR-list is not empty, AND_PROC receives messages from any of its active descendants. If a *fail* message or an "END" signal is received, the sending descendant is removed from the OR-list. If the current process is a Sync generator, a Sync signal with current process id will be sent out first before an answer is sent out. An answer is a list of variable bindings containing all the output variables in the goal. If an answer is received, the process sends the variable bindings to its proper consumers according to the channel list (procedure "output").

If the goal doesn't contain any input variables, the process simply spawns the OR processes and gathers the solutions. If all the OR descendants return *fail* , the AND process fails and a *fail* message is sent out through all the channels. Otherwise, the AND

† The END signal is always sent to the father no matter whether the father process is in the output channels or not.

process succeeds and an "END" signal is sent to all the channels via a *binding* message.

```
AND-PROC(goal,sync,id,channels);

[if_input_vars_in_goal(goal) → start(MERGE_AND(input_list));
        *[ input:=dequeue(queue);
           [input=END → send_to_all_channels(binding(END));
                        abort
           |input=SYNC# → send_to_all_channels(binding(SYNC#))
           |otherwise → newgoal:=bind_input_to_goal(goal,input);
                      [built_in(newgoal) → eval(newgoal,answer);
                                           output(answer)
                      |otherwise → OR_list:=[1..N]; i:=1;
                            % N is the number of clauses with the same head as goal
                               *[i<=N → OR[i]!spawn(OR_PROC(newgoal,i)); i:=i+1];
                               *[nonempty(OR_list) →
                                        *[OR[k]?binding(END) → remove(OR_list,k)
                                         |OR[k]?fail → remove(OR_list,k)
                                         |OR[k]?binding(answer) → output(answer)
                                         ]
                               ]
                      ]
           ]
        ]
|otherwise → [built_in_predicate(newgoal) → eval(newgoal,answer);
                                            output(answer)
            |otherwise → *[i≤N → OR[i]!spawn(OR_PROC(goal,i)); i:=i+1];
                         failed:=true;
                         *[nonempty(OR_list) →
                                *[OR[k]?binding(END) → remove(OR_list,k)
                                 |OR[k]?fail → remove(OR_list,k)
                                 |OR[k]?binding(answer) → failed:=false;
                                                          output(answer)
                                 ]
                         ]
            ];
            [failed → send_thrgh_channels(fail)
            | ¬ failed → send_thrgh_channels(binding(END))
            ]
].

procedure  output(answer)

[sync → send_to_all_channels(binding(Sync(id)))
|otherwise → skip];
send_each_channel(binding(answer)). % send each output binding through its channels.
```

Figure 3.5. The Program for AND-PROC

Let's assume each channel in our model has a finite *channel buffer* to store the messages that are not yet received by the receiver. A send operation succeeds as long as the buffer of the channel is not full. If the channel buffer is full, the sender suspends itself until the send operation is completed. A receive operation removes the first message in the channel buffer. When the channel buffer is empty, the receiver waits until some message is added to the buffer by a sender.

When MERGE_AND is created in an AND process, it allocates one *input buffer* for each input channel. The buffers are assumed to be large enough to store the whole input stream in one input channel. MERGE_AND merges the data in the input buffers according to the merge algorithm described in Chapter 5. When the next element of the input buffer is not available, MERGE_AND removes all the messages in the corresponding channel buffer and puts them into the input buffer, then resumes the merge operation.

Figure 3.6 shows the program for the routine OR_PROC. OR_PROC has two arguments: The first argument is the goal and the second argument is the process id, which also indicates the specific clause in the program to be unified with the goal in this OR process.

After OR_PROC is invoked, it first finds the clause to be unified with the goal ("find_the_id_th_clause") and then unifies the goal with the clause. If the unification fails, the OR process sends a *fail* message to its father and terminates. If the unification succeeds and the clause is a unit clause, then the process sends the binding generated in the unification followed by an "END" signal to its father and terminates. If the clause has an empty guard, a process list (UPL, which stands for Unfired Process List) containing all the literals in the clause body is created ("proc_list"), and an ordering algorithm is performed to construct the data flow graph of the literals in UPL ("ordering" and "refinement"). The ordered literals with annotated arguments are stored in FPL (which stands for Fired Process List); the producer and the consumers of each variable are stored in CT (the Channel Table). After the data flow graph is constructed, all the multiple paths in the graph are detected and the starting node of a multiple path is marked as a Sync generator ("mark_sync"). The above three steps, i.e., ordering, refinement, and mark_sync are combined into a so-called "ordering algorithm," which will be described in Chapter 4.

Now, OR_PROC is ready to spawn the AND descendants to execute each literal in the body. The OR process sends a *spawn* message to invoke each of its descendants. The message consists of a goal, a Sync attribute, an identifier (the above three items are stored in FPL[i]), and the channel information for the output variables in this goal. After all the descendants have been created, OR_PROC invokes MERGE_OR and starts to gather the answers from its descendants. If the OR process receives a *fail* message from any of its descendants, it is interrupted immediately. A *fail* message is sent to its father, a *kill* message is propagated to the other active descendants to kill all the ongoing computations in its subtree, and the process terminates. Upon receiving an "END" signal from any descendant, OR_PROC removes that descendant from its active process list (FPL). As long as the active process list is not empty and no descendant fails, OR_PROC keeps receiving messages from its descendants and puts the answers into the input buffers, ("put_input_buffer"). The answers in the input buffers are merged by MERGE_OR and the merged results are sent up to the father process.

If the guard is not empty, OR_PROC first performs the ordering algorithm to order the literals in the guard and spawns one AND process for each guard literal. The procedures are the same as described above except that only a fail message or an "END" signal may be received from its descendants because the guard literals don't produce any output to the goal. When all the AND processes for guard literals terminate successfully, OR_PROC spawns the AND processes for the literals in the clause body as usual. If any one of the guard literals fails, the OR process fails immediately and the other guard literals are killed as well.

When MERGE_OR refers to an empty buffer entry in the middle of the merge operation, it waits until the buffer entry is filled by OR_PROC. The program for MERGE_OR is omitted here and a more thorough description of the merge algorithm can be found in Chapter 5.

```
OR-PROC(goal,id);

[Father?kill → AND[1..N]!kill; abort
|otherwise → find_the_id_th_clause(goal,id,clause);
             unify(goal,clause);
             [if_unify_fail → Father!fail; abort
             |otherwise → [if_unit_clause(clause) → Father!binding(bind);
                                                     Father!binding(END); abort
                          |if_guard_exist → guard_list(clause,UGL);
                                                 ordering(UGL,FGL,CT);
                                                 mark_sync(FGL,CT);
                                                M:=length(FGL);
                                                i:=1;
                                               *[ i≤M → channels:=out_channel(FGL[i],CT);
                                                       AND[i]!spawn(AND_PROC(FGL[i],channels));
                                                        i:=i+1
                                                ];
                                               *[nonempty(FGL) →
                                                        [AND[k]?fail → Father!fail; i:=1;
                                                             *[i≤M → [i=k → skip
                                                                      |i≠k → AND[i]!kill
                                                                      ]; i:=i+1];
                                                                abort
                                                        |AND[k]?binding(END) → remove(FPL,k)
                                                        ]
                                                ]
                          |otherwise → skip
                          ];
                          proc_list(clause,UPL);
                          ordering(UPL,FPL,CT);
                          refinement(FPL,CT);
                          mark_sync(FPL,CT);
                          N:=length(FPL);
                          i:=1;
                         *[i≤N → channels:=out_channel(FPL[i],CT);
                                 AND[i]!spawn(AND_PROC(FPL[i],channels));
                                 i:=i+1
                          ];
                          start(MERGE_OR(input_list));
                         *[nonempty(FPL) →
                              [AND[k]?fail → Father!fail; i:=1;
                                             *[i≤M → [i=k → skip
                                                     |i≠k → AND[i]!kill
                                                     ]; i:=i+1];
                                                  abort
                              |AND[k]?binding(END) → remove(FPL,k)
                              |AND[k]?binding(answers) → put_into_buf(answers);
                              ]
                           ]; Father!binding(END)
             ]
].
```

Figure 3.6. The Program for OR-PROC

3.4 Summary

The Sync Model for the parallel execution of CLP has been described in this chapter. In this model, two kinds of processes, AND processes and OR processes, are interconnected into a dynamic process tree to realize the AND/OR proof tree of the given goal. Communication channels between sibling AND processes are also constructed to model the data flow graph of the AND literals.

In the construction of the data flow graph, any node that is a starting node of a multiple path is marked as a Sync generator. A Sync generator generates Sync signals appended with its own process id to separate each of its outputs. A process with two or more input channels merges its input streams based on the Sync signals carried in the inputs.

The functions of the AND process and the OR process are described in four routines: AND_PROC, OR_PROC, MERGE_AND, and MERGE_OR. AND_PROC and MERGE_AND are concurrent routines in an AND process, so as OR_PROC and MERGE_OR in an OR process. MERGE_AND merges the inputs received from the horizontal channels of an AND process and puts the result into a queue. AND_PROC gets inputs from the queue and processes them one by one.

The main innovation of our Model is a multiple-solution data-driven model to solve a nondeterministic logic program in parallel. Our model exploits AND parallelism and OR parallelism inherent in a logic program and returns all the solutions without specific requests. The synchronization mechanism proposed in our model is the first in the literature to realize a data-driven model with AND parallelism.

Comparison with Related Works

There are many proposals for the parallel processing of logic programs. Some of them introduce concurrent constructs into the language to make new concurrent languages, such as Relational Language, Concurrent Prolog and Epilog. We have discussed the difference between CLP and all those languages in the previous chapter. Some of the works propose various concurrent machine architectures to implement either AND parallelism, OR parallelism, or both. In the following, we compare our Sync Model with some of the related works in the second approach.

Conery's AND/OR Process Model [11] is very close to our Sync Model. Conery's Model also realizes AND parallelism and OR parallelism in a deterministic or a nondeterministic logic program. Our model is more efficient than Conery's in several ways.

(1). In Conery's Model, the unification of a goal with a set of clauses whose heads match the goal is done sequentially in a process, while in our model, the unification is done in parallel.

(2). In Conery's Model, the ordering algorithm is applied every time a new binding is generated, while in our model the ordering algorithm is performed only at the creation of the AND processes. As we shall see in Chapter 6, our model does a simple type checking when a new binding is generated and the data flow graph is changed by adding "dynamic links" for newly generated variables.

(3). Conery's Model handles AND parallelism by gradually reducing the data flow graph until it is empty. The graph reduction procedure needs a centralized control at the father process and the binding environment is also maintained at the father process. In our model, a network of AND processes is constructed based on the data flow graph, and only local binding information is passed around the channels. Therefore, the communication overhead and the workload of the father process are reduced.

(4). Conery's Model is a demand-driven model in which the solutions are sent out only upon request. It is awkward in handling nondeterministic programs. A complex backward execution algorithm is designed to trace back the data flow graph and redo the predecessors when a process fails. In our data-driven model, a merge algorithm using synchronization signals handles multiple solutions along communication channels. The control is simpler and the communication overhead is lower.

Most other models realize OR-parallelism in a shared-memory multiprocessor environment. Ciepielewski and Haridi's OR-parallel Token Machine is such an example [7], [8]. The major concern in their model is how to organize the local and global environment of each processor so that the memory can be utilized more efficiently. Bic [2] proposes a totally different dataflow architecture that represents a logic program as dataflow graphs. In the dataflow graph, arguments in a unit clause are represented by nodes, and relations between nodes are represented by arcs. The maximum parallelism is determined by the number of unit clauses in the program; therefore, it is suitable for database applications. Besides, AND-parallelism is restricted to the parallel execution of the AND literals without shared variables. Only a small portion of the previously proposed models falls into the category of data-driven models, which returns multiple solutions without backtracking, such as Lindstrom's stream-based model [28, 29], and Nakagawa's divided assertion set model [35]. Lindstrom's model exploits only OR parallelism with AND processes linearly connected.

Chapter 4

The Ordering Algorithm

4.1 Introduction

AND parallelism is difficult to implement due to potential binding conflicts of the shared variables between AND literals. By requiring that each shared variable has exactly one producer, we can eliminate binding conflicts. A data flow graph of all the AND literals in the clause body is then constructed such that a node represents an AND literal, and an edge is directed from the producer of a shared variable to a consumer of that variable. To construct the data flow graph of AND literals, an Ordering Algorithm is applied in each OR process. The data flow graph is represented in two ways: via variable annotations in the literals and via a channel table containing the producer and consumer information of shared variables. Maximum AND parallelism can be achieved in this data flow model, and efficient execution can be assured by avoiding superfluous computations due to unavailability of shared variables.

Since an AND process is created to handle an AND literal and a literal is represented by a node in the data flow graph, we use the terms *literal*, *process* and *node* as synonyms. Similarly, the terms *channel* and *link* are used interchangeably.

In the rest of this chapter, the ordering algorithm is presented in Section 4.2. Two examples are used to explain the algorithm in Section 4.3 and the complexity of the algorithm is analyzed in Section 4.4. Section 4.5 summarizes the chapter.

4.2 Ordering Algorithm

After an OR process successfully unifies its goal with a clause through a unifier θ, it replaces all the variables in the clause body by the substitution values in θ as long as substitution pairs exist for those variables. Afterwards, an Ordering Algorithm is performed to build up the data flow graph of the AND literals. The Ordering Algorithm contains three major steps: (1) the construction of the data flow graph, (2) the refinement of the graph, and (3) the marking of the Sync generators. These three steps correspond to the three procedures in Figure 3.7: ***ordering***, ***refinement***, and ***mark_sync***. In the first step, variable annotations are used to determine the modes (input or output) of all the uninstantiated variables in the AND literals. The ordering algorithm starts with an *Undecided Process List* (UPL) containing all the AND literals in the clause body. The algorithm determines the producer and the consumers of all the variables in the AND literals. It thus adds annotations to all the variables and moves the literals to a *Fired Process List* (FPL). A Channel Table (CT) is also constructed to store the producer and consumers' information of all the variables. Moreover, the literals are renumbered during this step so that their numerical order is consistent with their topological order in the data flow graph. In the second step, the data flow graph is further refined by creating "selective channels" and "True/False channels" for the literals that generate no output variables. As we shall see, this step is necessary to exploit the parallelism implied by the program so that a more efficient data flow graph can be constructed. In the third step, the algorithm searches for all the multiple paths in the data flow graph. If a multiple path is found, the algorithm marks the starting node of the multiple path as a Sync generator. The complete algorithm will be elaborated in the remainder of this section.

Data Structure:

The following data structures are used in the algorithm:

- UPL – a list of AND literal and identifier pairs that are not yet fired†.
- FPL – a list of fired AND literals with all their variable arguments annotated. Each entry in the list contains an AND literal with annotated arguments, a

† "A literal is fired" means that a literal is moved from UPL to FPL and all its variable arguments are annotated.

Sync attribute, and the identifier of the literal.

- CT – a table of triples (Var, Producer, Consumers-list), to record the producer and consumers of a variable.
- S – a stack containing distinct literals belonging to the paths starting from one specific literal.

Besides, the AND literals are initially identified 1 to N from left to right in the clause body with the goal of the current OR process numbered 0.

Algorithm:

Step 0: Initialization

CT:= ∅; FPL:= ∅;

UPL:= list of all literals.

Step 1: Construction of the data flow graph:

In this step, the producer and the consumers of each shared variable are chosen and the variables in each literal are annotated.

A literal can be fired if (1) all its input variables have a producer and the producers are already fired, and (2) the total number of output variables, input variables, and constant arguments of this literal is at least one. The first condition assures that the producer of a shared variable is always fired before the consumers of this variable. Therefore, the graph is guaranteed to be acyclic. The second condition is used to select a literal to be fired next among several candidates. The graph constructed with this simple rule is more efficient than the graph constructed without this rule. We may use different rules to select the next literal to be fired. For instance, the literal that has the maximal number of instantiated arguments is selected to be fired next. However, we need to scan all the unfired literals in order to find the next one to be fired. Hence, the complexity of the algorithm is increased from $O(n)$ to $O(n^2)$. Since the ordering algorithm is performed at run time, we decide to choose a simple rule for less overhead.

If none of the unfired literals satisfies the above conditions, the leftmost unfired literal in the clause body is chosen to be fired next.

```
a. forall $v_i$: $v_i \in$ uninstantiated variables in the goal
        add $\langle v_i, [], [0] \rangle$ into CT;

b. forall $l$: $l \in$UPL
   do   forall $v_i$: $v_i \in$ variable arguments in $l$
        do   if $v_i \notin$ CT $\rightarrow$ if $v_i$ is output annotated $\rightarrow$ add $\langle v_i, l, [] \rangle$ into CT
                               | otherwise $\rightarrow$ add $\langle v_i, [], [] \rangle$ into CT
                               fi
             | $v_i \in$ CT $\rightarrow$ if $v_i$ is output annotated $\rightarrow$ CT.$v_i$.*producer* := $l$
                               | otherwise $\rightarrow$ skip
                               fi
             fi
        od
   od ;
c. *index* := 1;
   while UPL$\neq \emptyset$
   do   *fired* :=false;
        forall $l$: $l \in$UPL
        do   forall $v_i$: $v_i$ is unannotated $\wedge$ CT.$v_i$.*producer* $\neq$[]
                  mark $v_i$ as an input variable in UPL;
             $b$ :=true;
             forall $v_i$: $v_i$ is an input variable in $l$
             do   $x$ :=CT.$v_i$.*producer*;
                  $b := b \wedge (x \neq [] \wedge x > N)$
             od   {*$b = \forall v_i : v_i$ is an input variable in l: $v_i$ has a producer and the*
                  *producer is fired*}
             if b$\wedge$(#constant arguments+#output variables+#input variables>0)$\rightarrow$
                  {*beginning of firing process*}
                  *newid* := *index* + $N$;
                  forall $v_i$: $v_i \in$ variable arguments in $l$
                  do     if $v_i$ is input $\rightarrow$ add *newid* into CT.$v_i$.*consumer*
                         |$v_i$ is unannotated$\vee v_i$ is output $\rightarrow$ CT.$v_i$.*producer* := *newid*;
                                             mark $v_i$ as an output variable in UPL
                         fi;
                  od
                  UPL:=UPL$-l$;
                  FPL[*index*] := $l$;
                  *index* := *index* + 1;
                  *fired* :=true
                  {*end of firing process*}
             |otherwise$\rightarrow$ skip
             fi
        od ;
```

```
d.      if $\neg fired \rightarrow l$ :=UPL[1];
                      do "firing process"
        |otherwise→skip
        fi
    od ;
e.  forall $v_i$: $v_i \in$ CT
    do  if CT.$v_i$.consumer = [] → CT:=CT−$v_i$
        |otherwise → skip
        fi
    od .
```

Step 2: Refinement of the Graph

In the data flow graph constructed in Step 1, a literal may have no output channels, i.e., no output variables. Such a literal is used as a predicate to verify if the inputs it received are valid or not. If the literal is proved true with a set of input bindings, this set of inputs is valid. Otherwise, the inputs are not valid. The OR process needs to know the results of this kind of literals so that it can make choices among the partial solutions it received based on the True/False values of those literals.

For instance, a data flow graph is constructed in Figure 4.1 for the clause:

$$g(Y) : -a(X), b(X), c(X,Y).$$

This graph is wrong because literal b makes selections on X while goal g is not notified. We have to rule out all the values of Y which are derived from some X values that make b false. There are two ways to fix the graph: First, we add a link from b to g to transmit the values of X that make b true (Figure 4.2.a). With this extra link, a becomes a Sync generator, and p will make proper choices on Y based on the input stream of X. Second, since only the X values that make b true are desirable, we can remove the link (a,c) and add a new link (b,c) from Figure 4.1 such that the output of a, X, will first be selected by b, then sent to c to produce Y. The new data flow graph is shown in Figure 4.2.b. We seem to lose parallelism in the second approach by serializing literals b and c. Nevertheless, the second approach is more efficient than the first one because unnecessary computations are avoided in process c, and the merge operation is eliminated in process g.

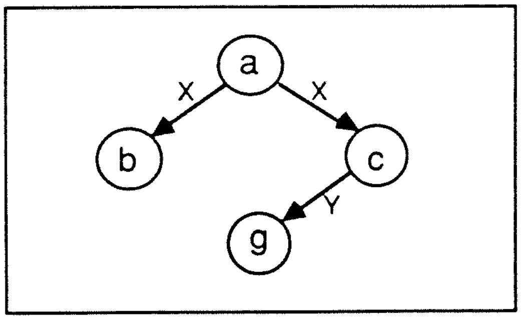

Figure 4.1. An Example with True/False Channels

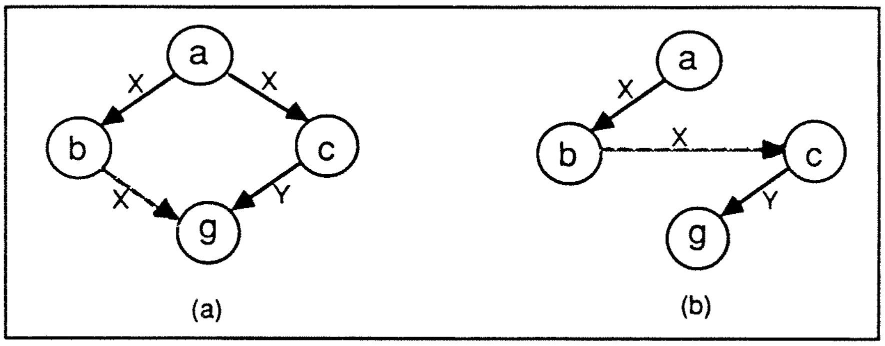

Figure 4.2. Two Refinements of Figure 4.1

Let's call the link (b, c) a ***selective channel***, for it selects and transmits the right values of X which make b true. If we can't find such channels for a literal with no output variables, then we need to add a ***True/False channel*** from it to the OR process to notify the OR process of its outcomes. Therefore, this step refines the data flow graph derived in Step 1 by including the selective channels and the True/False channels for all the literals with no output variables.

The refined data flow graph has to be acyclic. In other words, the insertion of selective channels should not introduce cycles in the graph. To assure the acyclicity of the graph, we only add selective channels such that the receiver of the channel is fired after all the antecedents of the sender. The antecedents of a literal are the producers of all the input variables of the literal. The numerical order of the fired literals is consistent with their firing order in Step 1, which implies the partial order in the data flow graph.

```
forall $l$: $l \in$ FPL $\wedge$ $l$ has no output variables
do     $new := false$;
       $prod := \emptyset$;
       forall $v_i$: $v_i$ is an input variable of $l$
              add CT.$v_i$.$producer$ into $prod$;
       forall $v_i$: $v_i$ is an input variable of $l$
       do     $c := CT.v_i.consumer$;
```

```
                $cl := \{c_i | c_i \in c : (\forall p_j \in prod : c_i > p_j) \wedge c_i \neq l\}$;
                if $l \in c \wedge cl \neq \emptyset \rightarrow$ add $\langle \overline{v_i}, l, cl \rangle$ into CT;
                                 CT.$v_i$.$consumer$ := $c - cl$;
                                 $new$ :=true
                |otherwise $\rightarrow$ skip
                if
            od ;
            if $\neg new \rightarrow$ add $\langle t/f, l, [0] \rangle$ into CT
            |otherwise $\rightarrow$ skip
            fi
        od .
```

Step 3: Marking of the Sync generators (Detection of the multiple paths):

A stack is built for each literal l in FPL that has more than one output channel. The descendants of l are pushed into the stack if they are not yet in the stack. This pushing process continues until either all the descendants of l are in the stack or a literal to be added to the stack is found to be already in the stack. If in the last case, l is marked as a Sync generator.

```
    forall $l : l \in$FPL $\wedge$ #consumers($l$) > 1
    do    $pt := 1$; $S := [l]$;
          while $S[pt] \neq \emptyset$
          do    $p := S[pt]$;
                forall $v_i : v_i$ is an output variable in $p$
                      forall $c_i : c_i \in CT.v_i.consumer$
                      do      if $c_i \notin S \rightarrow$ push $c_i$ into $S$
                              |$c_i \in S \rightarrow$ set Sync attribute of $p$ to true in FPL; goto $a$
                              fi
                      od
                od ;
                $pt := pt + 1$
          od
    $a$: od
```

The above algorithm always generates an acyclic data flow graph with one producer per shared variable. The ordering algorithm is correct for the following reasons:

1. The ordering algorithm selects exactly one producer for each variable.
2. The data flow graph generated in Step 1 is acyclic because a literal can be fired only when all the producers of its input variables have been fired (b is true in Step 1.c).

Therefore, the producer of a given variable is always fired before all the consumers of that variable. The firing order of the literals implies their partial order in the data flow graph; thus, the graph has no cycles.

3. The refined data flow graph generated in Step 2 is acyclic because the redirected links won't cause cycles in the refined graph. If a cycle were found in the refined graph, it would contain at least one redirected link, say (l_i, l_j). Let the cycle be $(l_i, l_j, \ldots, l_k, l_i)$, then l_k is the producer of an input variable of l_i and $l_j > l_k$ because a path exists from l_j to l_k. In Step 2, such a link (l_i, l_j) is never generated because l_j is excluded from *cl*. Therefore, the refined graph is also acyclic.

4.3 Examples

We now present two examples to illustrate how the ordering algorithm works. The first example is a quicksort program to sort a list of integers in nondescending order. This program is an example of the divide-and-conquer type applications. The second example is a database query program, which is a typical example for nondeterminism.

Example 1. quicksort

```
sort([X|Unsorted],Sorted) :- split(X,Unsorted?,Smaller,Larger),
                             sort(Smaller?,Sorted1),
                             sort(Larger?,Sorted2),
                             append(Sorted1?,[X|Sorted2],Sorted).
sort([],[]).

split(X,[A|L],[A|Smaller],Larger) :- A<X → split(X,L?,Smaller,Larger).
split(X,[A|L],Smaller,[A|Larger]) :- A≥X → split(X,L?,Smaller,Larger).
split(X,[],[],[]).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

:- sort([2,1,3],L).
```

Figure 4.3. Quicksort

At the beginning, a root node is constructed for the goal *sort*$([2,1,3], L)$. The root node is actually an AND node without sibling nodes. It then spawns two OR processes to do the unification for the two *sort* clauses in the definition. The second OR process fails to unify with the goal. The one that succeeded in unification has the clause as in Figure 4.4.

The number to the right of each AND literal is the identifier of that literal.

```
sort([2,1,3],L) :- split(2,[1,3],Smaller,Larger),          (1)
                    sort(Smaller?,Sorted1),                  (2)
                     sort(Larger?,Sorted2),                  (3)
                     append(Sorted1?,[2|Sorted2],L).         (4)
```

Figure 4.4. The Clause where the Ordering Algorithm Is to Be Applied

It is time to run the ordering algorithm to construct the Channel Table for all the shared variables.

1. Construction of the data flow graph:

    a. Initially, UPL contains four literals with identifier (1) to (4). Since there is one uninstantiated variable L in the goal, (L,[],[0]) is added into CT.

    b. Check all the literals in UPL and add (Smaller,[],[]), (Larger,[], []), (Sorted1,[],[]), and (Sorted2,[],[]) into the Channel Table. We now have a Channel Table with five variables.

    c. Since literal (1) has two constant arguments but no input variables (i.e., *b*=true), it will be fired first. The new identifier, *newid*, of literal (1) is 5 ($N = 4$ and $index = 1$). In the firing process, variables *Smaller* and *Larger* are annotated as output variables and "5" is put into the producer fields of *Smaller* and *Larger*. Then the literal is moved from UPL to FPL. Literal (2) has one input variable, *Smaller*, that has a producer (5) which is already fired. Therefore, literal (2) is fired next and *Sorted*1 is annotated as an output variable. The new identifier of (2) is (6), which is put into the consumer field of *Smaller* and the producer field of *Sorted*1 in CT. Afterwards, literal (3) and (4) are fired consecutively and they are given new identifiers (7) and (8), respectively. After all the literals are moved into FPL, the final CT is shown in Table 4.5 and the data flow graph is shown in Figure 4.6. In the data flow graph, the number in each circle is the original identifier of each literal. The new identifiers, which imply the firing order of the literals, are shown next to the circles.

2. Refinement of the graph:

    In this example, every literal has at least one output variable; thus, this step is omitted.

Variable	Producer	Consumers
L	8	0
Smaller	5	6
Larger	5	7
Sorted1	6	8
Sorted2	7	8

Table 4.5. Channel Table (CT) after Step (1)

3. Marking of the Sync generators:

   In order to detect possible multiple paths in the data flow graph as shown in Figure 4.6, a stack is built up for each literal that has more than one consumer for its output variables. Since literal (5) has two output variables, *Smaller* and *Larger*, a stack is constructed with (5) in the bottom. The two consumers of (5), (6) and (7) are then pushed into the stack and the pointer *pt* points to (6). Literal (6) has one output variable, *Sorted*1, with consumer (8). Since (8) is not in the stack, it is pushed into the stack and the pointer is incremented to (7). Literal (7) also has one output variable, *Sorted*2, with one consumer (8). Since (8) is already in the stack, a multiple path between literal (5) and (8) is found. Therefore, literal (5) is marked as a Sync generator (i.e., the Sync attribute of (5) is set to true in FPL). Repeat for literal (6), (7) and (8); no further multiple paths are found.

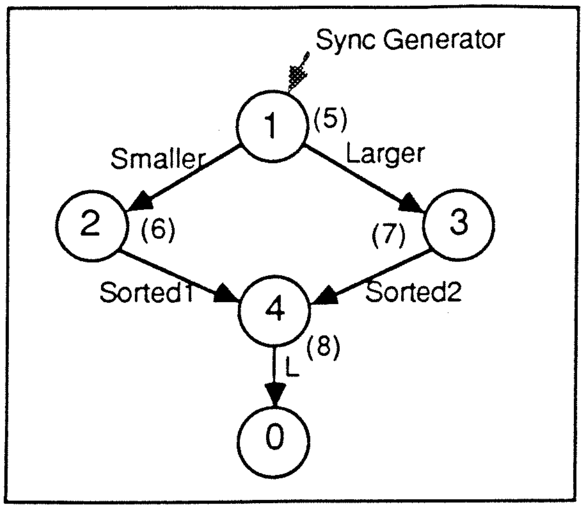

Figure 4.6. Data Flow Graph of *quicksort*

This program is deterministic. All the shared variables and the output variable have unique solutions. *Smaller*, *Larger*, *Unsorted* are stream variables. For more discussion of stream variables, please see Chapter 6.

Example 2: A Database Query

Figure 4.7 is a program to solve the query: "Is there a student such that a professor teaches him two different courses in the same room?" for a data base of students who take courses, professors who teach courses and courses held on certain weekdays and rooms. [37]

```
query(S,P):- student(S,C1),
             course(C1,D1,R),
             professor(P,C1),
             student(S,C2),
             C1≠C2,
             course(C2,D2,R),
             professor(P,C2).

% data base
student(robert,prolog).
student(john,music).
student(john,prolog).
student(john,math).
student(mary,physics).
student(mary,math).

professor(luis,prolog).
professor(luis,math).
professor(antonio,prolog).
professor(eureka,music).
professor(eureka,physics).

course(prolog,monday,room1).
course(prolog,friday,room1).
course(music,tuesday,room2).
course(math,tuesday,room1).
course(math,friday,room2).
course(physics,thursday,room3).
course(physics,wednesday,room2).

% query
:-query(S,P).
```

Figure 4.7. A Database of Students, Courses and Professors

The root node evaluates the goal *query(S,P)*. It spawns one OR process to do the unification and this OR process has the clause in Figure 4.8.

Now we should run the ordering algorithms against the seven AND literals in the clause

```
query(S,P):- student(S,C1),                    (1)
             course(C1,D1,R),                  (2)
             professor(P,C1),                  (3)
             student(S,C2),                    (4)
             C1≠C2,                            (5)
             course(C2,D2,R),                  (6)
             professor(P,C2).                  (7)
```

Figure 4.8. A Query Clause upon Which the Ordering Algorithm Is to Be Applied

body.

1. Construction of the data flow graph:
   After the initiation steps (1.a) and (1.b), the Channel Table contains seven variables: S, P, C1, D1, R, C2, and D2. All the producer and the consumer fields of the seven variables are empty except variable S and P which have [0] in their consumer fields.

   Since none of the variables in CT has a producer and none of the literals in UPL has a constant argument, step (1.c) fails (*fired*=false). In step (1.d), the first literal in UPL, (1), is fired and the new identifier for (1) is (8). The two unannotated variables in (1), *S* and *C1*, are annotated as output variables in UPL and their producer fields in CT are filled with (8). Then, the literal is moved from UPL to FPL. When all the literals in UPL are moved into FPL, the CT has the final form:

Variable	Producer	Consumers
S	8	0,11
P	10	0,14
C1	8	9,10,12
D1	9	[]
R	9	13
C2	11	12,13,14
D2	13	[]

Table 4.9. Channel Table (CT) after Step 1

   Notice that variables *D1* and *D2* have no consumers. In other words, these two variables are neither shared by any other literal nor required by the father process; hence they are useless and can be deleted from CT (Step (1.e)).

   The data flow graph for these seven AND literals is shown in Figure 4.10. Again, the number in each circle represents the original identifier of each literal. The new identifiers for the literals are shown next to the circles.

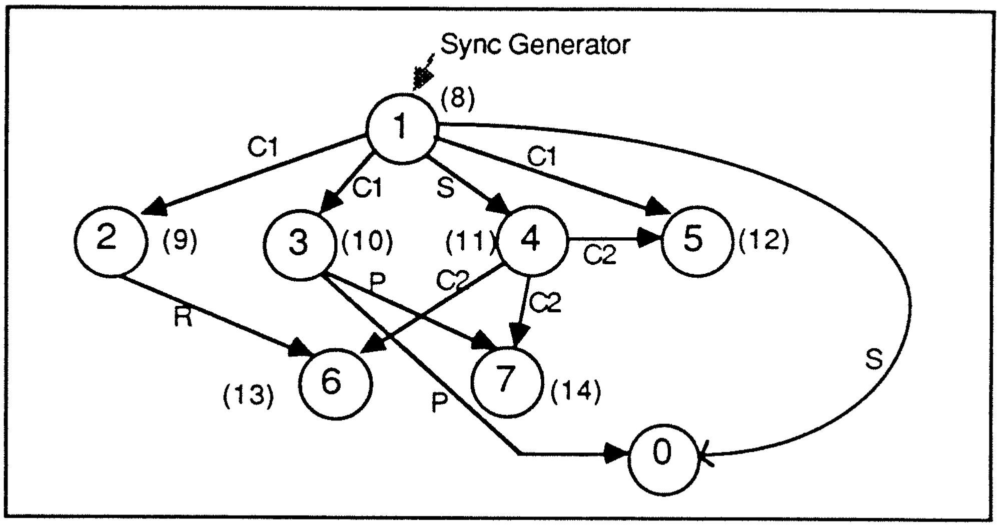

Figure 4.10. Data Flow Graph of $query(S, P)$

2. Refinement of the graph:

   In the above data flow graph (Figure 4.10), there are three literals without output variables, i.e., (12), (13) and (14). Literal (12) has two input variables $C1$ and $C2$, which have producers (8) and (11), respectively. Since the consumers of $C1$, (9) and (10), are both fired before the producer of $C2$, (11) ($cl = \emptyset$), no selective channel is created for $C1$. The two consumers of $C2$, (13) and (14), are fired after the producers of $C1$ and $C2$ ($cl = [13, 14]$); therefore, they are deleted from the consumer list of $C2$ and a new entry $(\overline{C2}, 12, [13, 14])$ is added into CT. Now CT looks like:

Variable	Producer	Consumers
S	8	0,11
P	10	0,14
C1	8	9,10,12
R	9	13
C2	11	12
$\overline{C2}$	12	13,14

Table 4.11. Channel Table (CT) after Refinement for Literal (12)

Literal (13) has three variables $C2, D2$ and R. There are two entries for $C2$ in CT, and (13) is found in the consumer list of the second entry, $\overline{C2}$. So, (14) is deleted from the consumer list of $\overline{C2}$ and a new entry $(\overline{C2}, 13, [14])$ is added into CT. No selective channel is constructed for variable R because (13) is the only consumer of R. $D2$ is a dummy variable that has been deleted from CT. For literal (14), (0) is deleted from the consumer list of P and a new entry $(\overline{P}, 14, [0])$ is added into CT. The final CT after this refinement step is shown in Table 4.12 and the refined data flow graph is shown in Figure 4.13.

Variable	Producer	Consumers
S	8	0,11
P	10	14
C1	8	9,10,12
R	9	13
C2	11	12
$\overline{C2}$	12	13
$\overline{C2}$	13	14
$\overline{P}$	14	0

Table 4.12. The Refined Channel Table (CT)

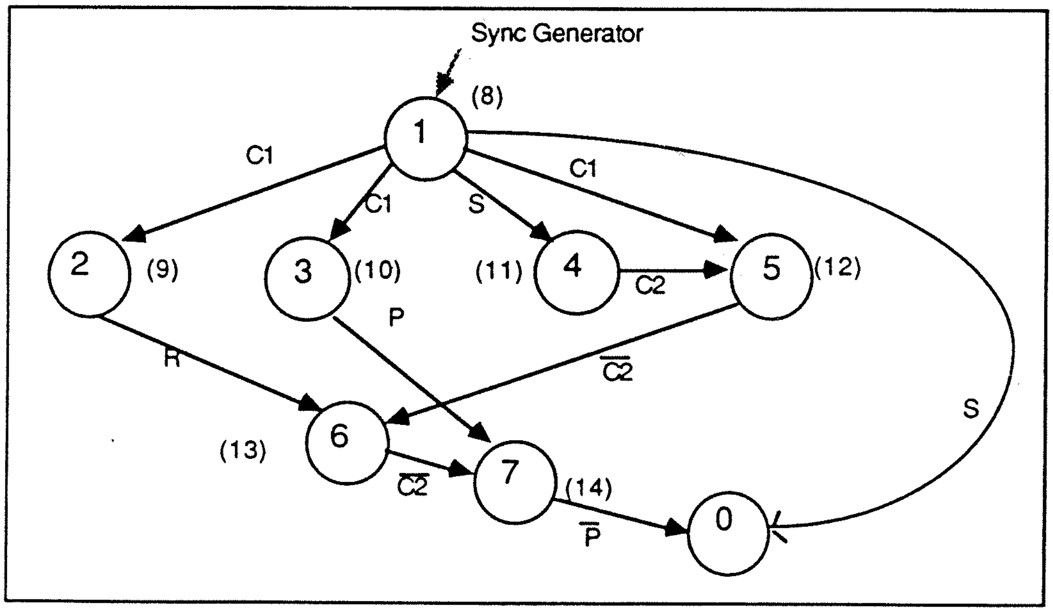

Figure 4.13. The Refined Data Flow Graph

3. Marking of the Sync generators:

   To detect the existence of multiple paths between any two literals in the data flow graph, we scan through FPL and find that literal (8) has two output variables, S and $C1$. A stack is built up with (8) at the bottom. Literals (0), (11), (9), (10), (12) are then pushed into the stack and the pointer is set to (0). Then, skip literal (0) because it is the father node and increment pointer to (11). Literal (11) has one output variable $C2$ with a consumer (12). Since (12) is already in the stack, a multiple path is detected between literal (8) and (12). Literal (8) is marked as a Sync generator in FPL. Repeating the same operation for literal (9) to (14), we find no more Sync generators.

4.4 The Complexity of the Ordering Algorithm

The complexity of the ordering algorithm can be determined by analyzing the complexity of each step in the ordering algorithm separately. Let's assume there are n literals in the process list (UPL) and m variables in the channel table (CT). The initiation step (Step 1.a) takes O(n) time to visit all the literals in UPL. In the firing step (Steps 1.b and 1.c), each entry of UPL may be visited more than once before it is moved to FPL. The total number of accesses to the table entries determines the complexity of the firing process. In the best case, the AND literals are well ordered according to its data flow graph; i.e., a variable's producer appears before all its consumers in the process list, and there is at least one constant argument in the first literal. Thus, each literal can be fired in the first access and the total number of accesses in this step is n. In the worst case, the AND literals share no common variables and none of their arguments are instantiated. The data flow graph is shown in Figure 4.14.a. In the process of firing an AND literal, all the unfired literals in UPL will be visited once, but none of them can be fired according to Step (1.b); then the first literal in the table becomes the next one to be fired according to Step (1.c). Therefore, to fire the first literal needs (n+1) table accesses and n accesses to fire the second literal, and so forth. The total number of accesses to complete the firing process is $(n+1)+n+\ldots+2=(n+3)n/2$. In another worst case example, the AND literals are reverse-ordered, and the data flow graph is a chain as shown in Figure 4.14.b. In order to fire an AND literal, all the unfired literals have to be accessed once because only the last literal in the table can be fired during each iteration. Therefore, the total count of accesses to fire all the AND literals is $n+(n-1)+(n-2)+\ldots+1=n(n+1)/2$.

In summary, the best-case complexity of the firing process is O(n) and the worst-case complexity is O(n^2). In most cases, the AND literals are well-ordered or mostly ordered, and the data flow graph is somewhere in between strictly parallel (Figure 4.14.a) and strictly sequential (Figure 4.14.b). On the other hand, we can always achieve O(n) complexity by inserting proper variable annotations into the program to guide the firing process. Table

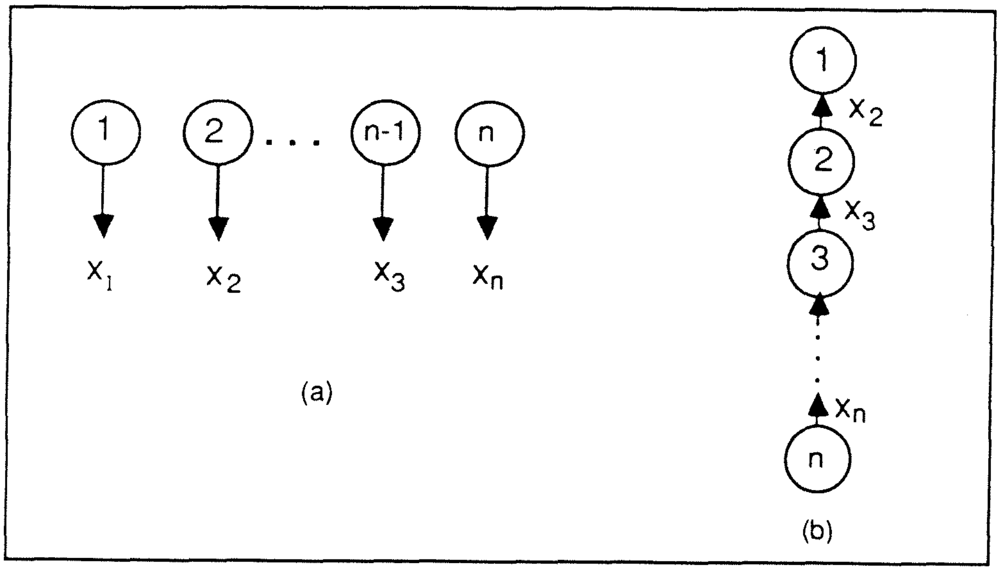

Figure 4.14. Two Worst Case Data Flow Graphs for the Firing Process

4.15 shows the simulation results of some examples. Examples 1 and 2 are the quicksort and query examples presented in the previous section. All the examples and simulation results can be found in Appendix A. Examples 1 and 4 have the best case performance; examples 2 and 3 need $2n$ accesses to fire all the AND literals because no one can be fired in the first visit of the whole table; i.e., (n+1) accesses are needed to fire the first literal. The fifth example has the AND literals in almost reversed-order, the complexity is about double the best case complexity, which is good compared to the worst case complexity. The last two examples show the two worst cases illustrated in Figure 4.14.

The complexity of the second step depends on the number of literals that have no output variables in FPL and the number of input variables in such literals. Let k be the number of literals that have no output variables and t be the number of input variables in such a literal. It takes O(n) steps to search FPL for this kind of literals. Then, for each input variable in such a literal, we modify one entry and possibly add a new entry into CT. This operation is repeated kt times. Since $k \ll n$ and t is considered as a constant (independent of n), the overall complexity of this step is O(n).

In the third step of the ordering algorithm, the multiple path starting from a given literal is detected by building up a stack for that literal and pushing all its descendants in the data flow graph into the stack until a common literal is encountered. The complexity

# nodes	Firing	Searching	# Sync Gen.
4	4	6	1
7	14	11	1
5	10	10	2
7	7	13	2
7	15	11	2
8	44	8	0
8	36	8	0

Table 4.15. The Simulation Results of the Ordering Algorithm

of this step is determined by the summation of the total number of literals in each stack. In search of a multiple path for a specific node, the subgraph of that node is traversed breadth-first until a common node is found. If there is no multiple path for that node, the whole subgraph is traversed once and the search fails. In our algorithm, a literal with only one consumer is ruled out immediately because it can't be a starting node of a multiple path. Therefore, the complexity can be computed as

$$\sum_{i=1}^{n}(\#\ \textit{of literals traversed in the subtree of node } i \textit{ before the first common literal is found})$$

and i is different from the identifiers of the literals that have only one consumer.

In the best case, each literal has at most one consumer. Thus, none of the literals can be a Sync generator and it takes O(n) time to detect. In the worst case, the AND literals are connected into a chain of three-node subtrees as shown in Figure 4.16. There is no multiple path in the graph. To determine whether the topmost node is a Sync generator, all the nodes in the graph will be traversed once, i.e., n nodes. Likewise, to detect the multiple path of any other node, all the nodes in the subtree of that node have to be traversed once. Therefore, the total number of nodes to be traversed is

$$n + 1 + (n-2) + 1 + (n-4) + \ldots + 1 + 3 + 1 + 1 = \frac{(n^2 + 4n - 1)}{4}$$

where n is an odd number.

In the average case, let's assume that the AND literals are connected into a complete binary tree and that no multiple path exists. Again, the complexity of detection of multiple

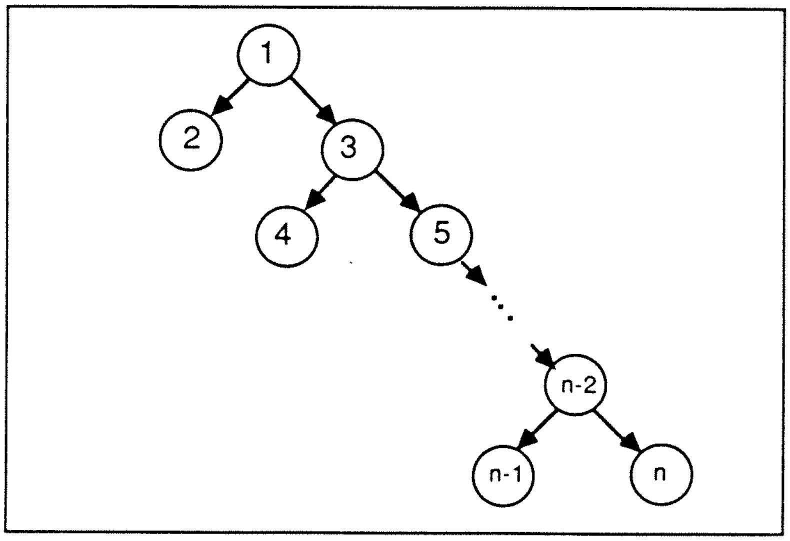

Figure 4.16. The Worst Case Data Flow Graph for Detection of Multiple Paths

paths is computed by summing up the number of nodes in the subtree of each node; i.e.,

$$\begin{aligned}
&n + 2 \times \frac{(n-1)}{2} + 4 \times \frac{(n-3)}{4} + 8 \times \frac{(n-7)}{8} + \ldots + \frac{n+1}{2} \times 1 \\
=&n + (n-1) + \ldots + (n - (2^i - 1)) + \ldots + (n - (2^{\lg n - 1} - 1)) \\
=&n \times \lg n - \sum_{i=1}^{\lg n - 1} (2^i - 1) \\
=&(\lg n - 1)n + \lg n \approx O(n \lg n).
\end{aligned}$$

If there exist multiple paths in the data flow graph, it usually takes less time to detect because the search stops when the first common node is found. Therefore, it is not necessary to traverse all the nodes in the subgraph. In summary, the detection of multiple paths has average complexity O($n \lg n$), best case complexity O(n) and worst case complexity O(n^2). The third column of Table 4.15 shows the total number of nodes traversed in this step for various examples, and the last column of the table shows how many Sync generators there are in each example. In these examples, the complexities are always below $n \lg n$.

The complexity of the third step is proportional to the number of Sync generators, say k. When $k = 1$, this step is skipped. For $k > 1$, sorting of k numbers takes $O(k \lg k)$ time on average. Then, it takes n steps to scan through FPL to change the identifiers of the Sync generators into the new identifiers. In the meantime, the identifiers in CT need to be changed accordingly. Let t be the total number of input and output variables of a Sync generator and k' be the number of Sync generators whose identifiers are changed after reordering; then $k't$ is the number of entries needed to be changed in CT. Therefore, the overall complexity of step (3) is $O(k \lg k) + O(n) + O(k't) \approx O(n)$, where t is a constant, $k' \leq k$ and $k \ll n$.

In conclusion, Step (1) and Step (3) of the ordering algorithm have the same complexities for the best and the worst cases; therefore, the best and the worst case complexity of the ordering algorithm are O(n) and O(n^2), respectively. The average complexity of the ordering algorithm is determined by step 2, i.e., O($n \lg n$). The simulation results of the ordering algorithm and the Prolog program are attached in Appendix A.

4.5 Summary

In this chapter, we presented the ordering algorithm in our Sync Model. The ordering algorithm is applied in an OR process to construct the data flow graphs of the AND literals in the clause body when unification succeeds. Besides constructing the data flow graph, the ordering algorithm also detects the multiple paths in the graph and marks Sync generators. The average complexity of the ordering algorithm is O($n \lg n$) with n AND literals. In most cases, the AND literals in a clause body are well-ordered; therefore, a linear complexity can be achieved.

In the literature, several different mechanisms are proposed to handle AND parallelism. Among them, static or dynamic join techniques [13] appear to be very inefficient. The forwarding technique [13, 28] ignores the potential parallelism by chaining the sibling AND processes into a pipeline and forwarding the solutions from the left to the right of the chain. The data flow approach is adopted mostly by language designers. In many concurrent languages [9, 45, 54], programmers add variable annotations to specify the data flow between literals; thus the data flow graph is determined by the programmers. A different approach toward the data flow model is to let the data flow graph be constructed by the system. Conery's AND/OR process model [11] constructs the data flow graph dynamically, and Chang and DeGroot suggest an algorithm to construct the graph statically [6]. In our Model, we take both approaches in the construction of data flow graph: by programmer as well as by system. Therefore, CLP can be used as a concurrent language, and a programmer has freedom to control the execution of the program. On the other hand, the Sync Model is capable of executing a logic program concurrently and efficiently by performing the ordering algorithm just described.

Our ordering algorithm differs from Conery's algorithm [11] in the following aspects: Conery's algorithm has to be performed whenever a new variable binding is produced, while our algorithm is performed only when the AND processes are created after a successful unification. Therefore, our scheme is much more efficient than Conery's. Besides, Conery used mode declaration to determine the flow directions of the variables, while we use variable annotations. Variable annotation is more flexible than mode declaration. The former provides local and specific information for each clause definition, whereas the latter can give only global information for one predicate. An exemplary case is that one variable in a predicate can be used as an input variable in one place and an output variable in another place. With variable annotations, we can put different annotations in different places to indicate how the variable is used. With mode declarations, this situation cannot appropriately be dealt with. Another advantage of our algorithm is that the refinement step exclusive for our algorithm exposes the real parallelism implied by the program so that a more efficient data flow graph can be constructed. Lastly, our algorithm has extra steps to detect multiple paths in the data flow graph, which is not necessary in Conery's model.

Chang and DeGroot use a static data dependency analysis to construct a set of possible data flow graphs for the AND literals [6, 14]. In cooperation with a run-time type checking, a data flow graph can be constructed to implement AND parallelism. Our algorithm is called after each procedure call (i.e., a successful unification) to construct a unique data flow graph. As we shall see in Chapter 6, the data flow graph may be modified dynamically due to the partially instantiated terms in the result. The detection of partially instantiated terms is similar to but simpler than the type checking algorithm in [14].

Chapter 5

The Merge Algorithm

5.1 Introduction

In the Sync Model, a process has to handle multiple input streams from different sources. For example, an OR process has to merge all the partial solutions from its AND descendants to form its solutions, and an AND process has to merge the input streams from other sibling AND processes to form input combinations to itself. It is particularly true for a nondeterministic program, in which multiple partial solutions may be generated, transmitted and validated by different processes. A merge algorithm that synchronizes the execution of all the cooperating processes is the key to the success of our Sync Model.

The merge algorithm in an AND process is basically the same as the one in an OR process. The only difference is that the input stream of the latter one may contain True/False values instead of variable bindings. In the following, the merge algorithm refers to the one in an AND process.

The merge algorithm operates only when there exist two or more input variables in a process. An input stream consists of Sync signals, variable bindings, and an END signal at the end. A variable binding is a pair containing a variable name and its binding value. The Sync signal carries the process identifier that identifies the generator of the Sync signal. Sync signals are nested when the receiving node belongs to two or more different multiple paths. In essence, the merge algorithm forms a Cartesian Product over the input streams to form all possible input combinations. When Sync signals appear, the algorithm forms Cartesian Products over parts of the input streams separated by pairs of identical Sync signals. In other words, only the input elements in between the corresponding pair of Sync signals can be combined, and the input streams are thus synchronized by the Sync signals.

In the rest of this chapter, the base-case algorithm (i.e., no input stream contains Sync signals) is first described in the next section. The Cartesian Product implemented as nested loops is first presented. Such a simple implementation is inefficient because the process may keep waiting for the inputs from a slow channel. A modified algorithm is then given. The new algorithm reduces the waiting time by forming the Cartesian Product over the available portions of input streams while the rest of the inputs are not yet there. In other words, the merge operation is concurrently processed with the receiving of the inputs in each input stream. The general algorithm with input streams containing Sync signals is presented in Section 5.3. The general algorithm is a recursive algorithm that recursively peels off Sync signals in two streams and finally forms the Cartesian Product over the data inputs enclosed by the innermost Sync signal pair using the base-case algorithm. The algorithm for n streams can be derived by generalizing the two-stream algorithm. In the last section, a correctness proof for the n-input general merge algorithm is presented.

Throughout the algorithms, $buf[i,j]$ is used to represent the j-th input in the i-th input buffer, where $1 \leq i \leq n$ and n is the total number of input buffers. Each buffer is assumed to have enough capacity to store the whole input stream. $Index[i]$ points to the position that is currently being merged and $avail[i]$ points to the top of the available portion of buffer i. Procedure *put(entry)* adds a new element *entry* into the output queue, where *entry* can be a Sync signal, an array of n input bindings or an "END" signal.

5.2 Base-case Algorithm

The straightforward implementation of a Cartesian Product over n buffers is shown in Figure 5.1. Notice that the input stream in each buffer is ended with a special symbol "END". When an input buffer has no more inputs available and the "END" is not yet detected, the process simply waits until the next input is received and then proceeds.

Since the merge algorithm is operating concurrently with the receiving of inputs in each

```
 index[n]:=1;
{ busy waiting when the input is not available }
  *[  buf[n,index[n]]≠END  →
      index[n-1]:=1;
      *[ buf[n-1,index[n-1]]≠END  →
            .....
            index[1]:=1;
            *[  buf[1,index[1]]≠END  →
                  put((buf[1,index[1]],...,buf[n,index[n]]));
                  index[1]:=index[1]+1;
             ];
            .....
          index[n-1]:=index[n-1]+1;
       ];
      index[n]:=index[n]+1;
   ];
  put('END');
```

Figure 5.1. Base-case Algorithm: Simple Version

input buffer, the simple iterative loop implementation may be inefficient due to waiting for the inputs from a slow channel. A more efficient implementation is shown in Figure 5.2.

The revised algorithm repeatedly forms the CP (abbreviation for Cartesian Product) over the available portions of the n input streams. Whenever an input buffer receives more inputs, procedure *cp* is called repeatedly to form the CP over the newly received inputs and the available portions of the other input buffers. Then $avail[j]$ is advanced to the location of the newest available input. The algorithm repeats the above operations for each input buffer until all the input buffers are ended with "END". Figure 5.3 illustrates the outcome of procedure $cp(e, i)$. The shaded areas are the available portions of the input buffers in which the CP is already generated. Procedure $cp(e, i)$ generates the CP of $buf[i, index[i]]$ and the shaded areas of the rest of input buffers.

5.3 General Algorithm

If Sync signals appear in at least one input stream, the general merge algorithm applies. We first present the general algorithm for two input streams and later show how to generalize the algorithm to n input streams.

In the ordering algorithm, the literals have been renumbered so that their numerical order is compatible with their partial order in the data flow graph. The linear ordering

```
{ Global Variables}
integer n;  { number of input buffers}
integer array index[1:n], avail[1:n];  { pointers}
input buffer buf[1:n,1:m];  { n input buffers with length m which is large enough to contain
                                   the whole input streams}
buffer entry[1:n]; { a buffer to contain the next output}

{ Cartesian Product of the available portions of the n input buffers except the i-th buffer
  which is fixed to an element e}
procedure  cp(e,i);
begin
    entry[i]:=e;
    cp1(i,1)
end .

{ Cartesian Product over the available portions of buf[k] to buf[n] except buf[i]}
procedure  cp1(i,k);
begin
    [ k>n → put(entry)
    | k=i → cp1(i,k+1)
    | otherwise → l:=1;
                  *[ l≤avail[k] → entry[k]:=buf[k,l];
                                   cp1(i,k+1);
                                   l:=l+1
                   ]
    ]
end .

{ Main Program }
begin
    i:=1;
   *[ i≤n → index[i]:=1; avail[i]:=0; i:=i+1 ];
    i:=1;
   *[ ∃k: 1≤k≤n: buf[k,index[k]]≠'END' →
        *[ i≤n → *[ ¬empty(buf[i,index[i]])∧buf[i,index[i]]≠'END' →
                                                    cp(buf[i,index[i]],i);
                                                    index[i]:=index[i]+1
                   ];
                   avail[i]:=index[i]-1;
                   i:=i+1
         ]
    ]
end .
```

Figure 5.2. Base-case Algorithm: Modified Version

of the Sync signals in an input stream is always assured by the merge operation which performs an n-way merge on n input streams.

The general algorithm consists of two principal operations: merge on the same Sync

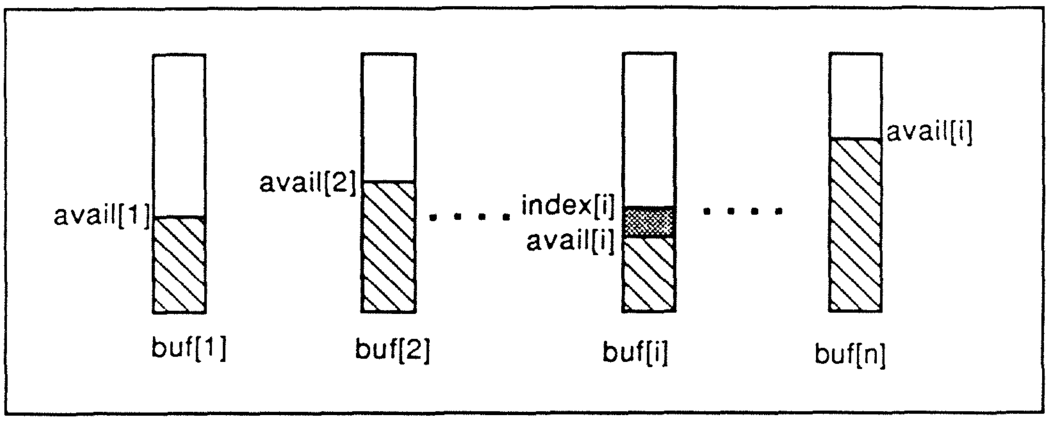

Figure 5.3. Procedure *cp(e,i)*

signals and merge on different Sync signals. First, let two input streams contain the same Sync, say S, and the two input streams are $A = (S, A_1, S, A_2, \ldots, S, A_n, \text{END})$ and $B = (S, B_1, S, B_2, \ldots, S, B_n, \text{END})$; then the merge result is a sequence of CP's over the corresponding portions of the two input sequences which are separated by a pair of consecutive S's, i.e.,

$$A \times B = (S, A_1 \times B_1, S, A_2 \times B_2, \ldots, S, A_n \times B_n, \text{END}) \tag{5.1}$$

where A_j stands for a sequence of data inputs, so as B_j for $1 \le j \le n$, and $A_j \times B_j$ stands for the CP of A_j and B_j.

The second principal operation handles the merge of two sequences with different Syncs. Let two input streams be $A = (S1, A_1, S1, A_2, \ldots, S1, A_n, \text{END})$ and $B = (S2, B_1, S2, B_2, \ldots, S2, B_m, \text{END})$, and let S1<S2 so that S1 becomes the outer Sync in the merge result. The linear ordering of the Sync signals appearing in a merged stream guarantees that the common Syncs appearing in two input streams are in the same order. Therefore, the merge algorithm works correctly. The merge result can be computed as follows:

$$\begin{aligned} A \times B &= (S1, A_1 \times B, S1, A_2 \times B, \ldots, S1, A_n \times B, \text{END}) \\ &= (S1, S2, A_1 \times B_1, S2, A_1 \times B_2, \ldots, S2, A_1 \times B_m, \\ &\quad S1, S2, A_2 \times B_1, S2, \ldots, A_2 \times B_m, \\ &\quad S1, S2, \ldots\ldots, S2, A_n \times B_m, \text{END.}) \end{aligned} \tag{5.2}$$

The merge result is actually the CP of all the data inputs of the two streams when the two input streams contain different Syncs. In order to maintain the synchronization information, we first form the CP's over the whole input stream B and a portion of stream A, i.e. A_i for all i and separate the CP's by S1. In each $A_i \times B$, again we form a set of CP's of $A_i \times B_j$ for all j and separate them by S2. Since the CP "$A_i \times B_j$" contains no Sync signals, the base-case algorithm is applied. In the result, the number n of Sync signals S1 is preserved, and the number of Sync signals S2 is increased to $n \times m$ because S2 is nested inside S1.

The general algorithm for two input streams is recursively defined on the two principal operations. The Sync sequences of the input streams are linearly ordered; i.e., the index of a Sync signal is larger than all the Syncs which are outer to it and smaller than all the Syncs inner to it. At each recursion step, the outermost Sync signals of the two input streams are compared. If they are the same, the first principal operation is called. If they are different, the second principal operation is called. The merge algorithm is called recursively to compute each $A_i \times B_i$ in Eq.(5.1) or each $A_i \times B$ in Eq.(5.2). When the merge algorithm is called to merge two input streams without any Sync signals, the base-case algorithm is applied yielding the CP. The merge result preserves the linear ordering of the Sync sequence. Figure 5.4 presents the major procedures of the merge algorithm: *merge*, and *scanto*. Procedure *merge* merges the input streams in $buf1$ and $buf2$, and puts the result in an output queue. $buf1$, $buf2$ are one-dimensional tables with sufficient size to store the input streams. Procedure *put* puts one merged result into the output queue. The boolean function *sync* determines whether the given argument is a Sync signal. Procedure *merge* has a guarded command with four alternatives: (1) neither of the inputs contains Sync signals: *cp* is called to derive the Cartesian Product; (2) either $buf1$ contains Sync signals and $buf2$ does not, or both inputs have Sync signals and the outermost Sync of $buf1$ is smaller than that of $buf2$: the second principal operation applies; (3) same condition as (2) with $buf1$ and $buf2$ switched: the second principal operation also applies with A and B switched; and (4) both inputs contain Sync signals and the outermost Syncs of the two inputs are the same: the first principal operation applies. Procedure *scanto* divides the input buffer into two parts by the first occurrence of some specific Sync signal S. Procedure *cp* is the base-case merge algorithm which generates the CP of the data elements in two buffers.

If there are more than two input buffers and some of them have one or more Sync signals, the above algorithms can be generalized easily. With n input streams in which each has an ordered Sync sequence, the merge algorithm applies recursively to remove the

```
procedure  merge(buf1,buf2)
begin
    [buf1=∅∨buf1="END"∨buf2=∅∨buf2="END"  → skip
    |otherwise→ A:=buf1[1]; B:=buf2[1];
        [¬sync(A)∧¬sync(B)  → cp(buf1,buf2)                           (1)
        |sync(A)∧(¬sync(B)∨(A<B))  → scanto(buf1,A,buf11,buf12);      (2)
                                     put(A);
                                     merge(buf11,buf2);
                                     merge(buf12,buf2)
        |sync(B)∧(¬sync(A)∨(B<A))  → scanto(buf2,B,buf21,buf22);      (3)
                                     put(B);
                                     merge(buf1,buf12);
                                     merge(buf1,buf22)
        |sync(A)∧sync(B)∧(A=B)  → scanto(buf1,A,buf11,buf12)          (4)
                                  scanto(buf2,B,buf21,buf22);
                                  put(A);
                                  merge(buf11,buf12);
                                  merge(buf12,buf22)
        ]
    ]
end  of procedure merge.

procedure  scanto(buf,S, buf1,buf2)
begin
     i:=2;
    *[ buf[i]≠S∧buf[i]≠"END"  → buf1[i]:=buf[i]; i:=i+1];
     j:=1; N:=length(buf);
    *[ i≤N  → buf2[j]:=buf[i]; i:=i+1; j:=j+1]
end  of procedure scanto.
```

Figure 5.4. General Algorithm for Two Buffers

smallest Sync signal of the n outermost ones of the input streams one at a time. When the smallest Sync is common to several input streams, all those Syncs will be removed at once. When none of the input streams contains Sync signals, the Cartesian Product over n input streams is performed. For instance, if $merge(buf1, buf2, \ldots, bufn)$ is called and a smallest Sync S is found in both $bufi$ and $bufj$, the following program is executed:

```
scanto(bufi,S,bufi1,bufi2);
scanto(bufj,S,bufj1,bufj2);
put(S);
merge(buf1,...,bufi1,...,bufj1,...,bufn);
merge(buf1,...,bufi2,...,bufj2,...,bufn).
```

The merge algorithm in an OR process merges the partial solutions received from its AND descendants to form all the legal solutions of this OR subtree. The partial solutions received from one AND descendant could be variable bindings or true/false values. The true/false values are used to select the merge result from other channels. If the value is true, the merge algorithm merges the partial solutions as usual. If the value is false, the merge algorithm skips the merge operation and returns false instead. In addition, the merge algorithm in an OR process eliminates all the Sync signals in the merge result so that the solution stream sent up to the father AND process contains no Sync signals.

5.4 Correctness Proof

The merge algorithm returns all the possible input combinations for a process with multiple inputs. We must assure that all the correct combinations are derived so that the correctness of our model is guaranteed. To prove the correctness of the merge algorithm, we need to prove three theorems.

Before presenting the theorems, we shall define the structure of an "input stream" syntactically and give a formal treatment of how an AND process transforms one or more input streams into an output stream.

Definition: An *input stream* $\Sigma_R(D)$ can be defined recursively:

1. $\Sigma_\emptyset(D) \equiv D$

2. $\Sigma_R(D) \equiv \Sigma_{R' \cup \{i\}}(D) \equiv \Sigma_{R'}(\Sigma_{\{i\}}(D)) \equiv \Sigma_{R'}((S_i, D_{v_R})^{n_i})$, where $\forall j \in R' : i > j$

where R is an ordered set of integers. Each element in R is a Sync that appears in the input stream. Let's call R the *Sync sequence* of this input stream. We slightly abuse notations and represent R by the array $R[i]$, s.t., $i < j \Rightarrow R[i] < R[j]$. Σ_R is an operator defined recursively over the input data, D, where D is the input stream with all the Syncs removed. Applying $\Sigma_{\{i\}}$ over D is to divide D into n_i groups and separate each group by a Sync S_i. Each group of input data, D_{v_R}, is called a *data segment*, which is uniquely identified by a *rank vector*, v_R. In (2), v_R is a vector of length $|R| = r$ where $|R'| = r - 1$ and $v[r] = k$ for $1 \leq k \leq n_i$. Therefore, D_{v_R} represents a data segment that is produced by the k-th output of the Sync generator S_i. Besides, $(S_i, D_{v_R})^{n_i}$ is a regular expression denoting the concatenation of the string (S_i, D_{v_R}) n_i times. Notice that the data segment D_{v_R} is changed every time the syntactic structure of the input stream is transformed. The above notation is used to represent the syntactic structure of an input stream. How D_{v_R} is changed by different transformations of the input stream will be explained later.

For instance, an input stream with two different Sync signals S_i and S_j, $i < j$, can be represented by

$$\Sigma_{\{i,j\}}(D) \equiv \Sigma_{\{i\}}(\Sigma_{\{j\}}(D)) \equiv (S_i, (S_j, D_{v_{\{i,j\}}})^{n_j})^{n_i}.$$

There are two ways to change the structure of an input stream in our model. First, if an AND process is a Sync generator, the structure of the output stream is derived by concatenating an extra Sync signal to the Sync sequence of the input stream. Second, if an AND process has more than one input, say n, the structure of the merge output can be derived by n-way merge of the n Sync sequences. Figure 5.5 shows the two possible transformations of an AND process given one input and one output. In Figure 5.5.a, the structures of the input and the output streams are the same because the AND process is not a Sync generator. In Figure 5.5.b, the AND process is a Sync generator which generates Sync S_i and the output stream has the structure $\Sigma_{R\cup\{i\}}$. Because of the total ordering of the Sync generators, i is guaranteed to be larger than any element in R. Figure 5.6 shows the input-output transformation of the merge algorithm, given n input streams. The Sync sequence of the output is derived by n-way merge of the n input Sync sequences. An AND process with n inputs and one output can be represented by one merge operation (Figure 5.6) followed by one of the two AND operations (Figure 5.5), depending on whether or not the AND process is a Sync generator. Being aware of the transformation of the input stream structures, we show how the data segments are changed during these two different kinds of operations. Without loss of generality, we assume $n = 2$ in the sequel.

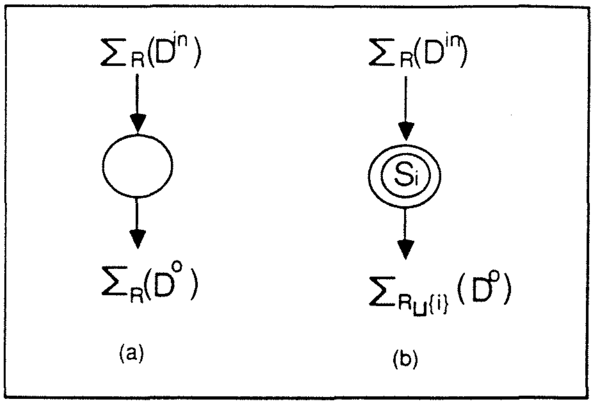

Figure 5.5. The Transformation of an AND Process with a Single Input

Definition: An *ordered union* operator "$\sqcup$" is defined as $R = R_1 \sqcup R_2$, where R, R_1 and R_2 are ordered sets (i.e., the elements in the set are sorted in ascending order), and

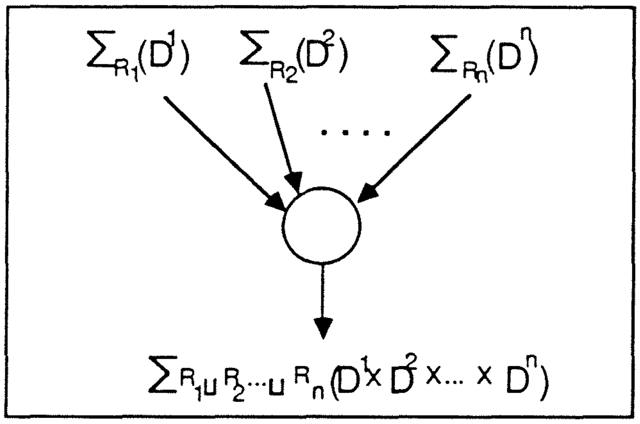
$\Sigma_{R_1}(D^1)$
$\Sigma_{R_2}(D^2)$
$\Sigma_{R_n}(D^n)$
....
$\Sigma_{R_1 \sqcup R_2 \cdots \sqcup R_n}(D^1 \times D^2 \times \ldots \times D^n)$

Figure 5.6. The Transformation of the Merge Algorithm with n Inputs

$R = R_1 \sqcup R_2$. In other words, it is equivalent to a two-way merge.

A rank vector v_R corresponding to a Sync sequence R is a vector of length $|R|$ and $v_R[i]$ is the rank of the Sync signal $R[i]$. Let each element of v_R, $v_R[i]$ be assigned a *name* $R[i]$, i.e., the corresponding Sync signal of $v_R[i]$. Since no two elements in R are the same, $v_R[i]$ can be uniquely identified by its name: $v_R.R[i] \equiv v_R[i]$.

Definition: An *ordered join* operator " $\underline{\sqcup}$ " is defined as $v_R = v_{R_1} \underline{\sqcup}\, v_{R_2}$, where $R = R_1 \sqcup R_2$, R, R_1, and R_2 are the corresponding Sync sequences of v_R, v_{R_1}, and v_{R_2}. v_R is the natural join of v_{R_1} and v_{R_2} on the elements with the same names. More precisely, let $R_1 \cap R_2 = \{A_i | 1 \leq i \leq k\}$; then $v_R = v_{R_1} \underline{\sqcup}\, v_{R_2}$ if and only if

1. $\forall i : 1 \leq i \leq k : v_{R_1}.A_i = v_{R_2}.A_i$
2. $v_R.A = \begin{cases} v_{R_1}.A, & \text{if } A \in R_1; \\ v_{R_2}.A, & \text{if } A \in R_2. \end{cases}$

Theorem 5.1. *Given a Sync generator S_i with one input and one output, and with the input stream having the form $\Sigma_R(D)$, then the output of the Sync generator is $\Sigma_{R \sqcup \{i\}}(D')$ and*

$$\bigcup_{\forall v_o, j : v_o[j] = v_i[j], 1 \leq j \leq |R|} D'_{v_o} = \bigcup_{\forall d \in D_{v_i}} f(d) \tag{5.3}$$

where f is the function to transfer an input to an output.

Proof: A Sync generator generates Sync signals to separate each output. Therefore, it changes the syntactic structure of the input stream by adding new Sync signals S_i's as the innermost Syncs. The new Sync is greater than any Sync signal in R since the numerical order of the Sync generators implies their partial order and since all the Sync generators that generate the Syncs in R cannot be preceded by S_i in the data flow graph. Each data segment D'_{v_o} contains at most one element that is generated by one input element in the input data segment D_{v_i} with $v_o[j] = v_i[j], \forall j : 1 \leq j \leq |R|$. In other words, the outputs generated by one input data segment D_{v_i}, the right-hand side of (5.3), are separated into several data segments by the Sync signal S_i's. Each of these data segments has a different value in the last element of its rank vector. The union of these data segments gives all the outputs generated by the corresponding input data segment. ∎

Theorem 5.2. *Given two input streams $\Sigma_{R_A}(D^a)$ and $\Sigma_{R_B}(D^b)$, the result generated by the merge algorithm is $\Sigma_{R_C}(D^c)$, where $R_C = R_A \sqcup R_B$ and each data segment $D^c_{v_c}$ in D^c is defined as a Cartesian Product of $D^a_{v_a}$ and $D^b_{v_b}$ such that*

$$D^c_{v_c} = D^a_{v_a} \times D^b_{v_b} \qquad \text{with} \quad v_c = v_a \sqcup\!\!\!\!\_ \; v_b. \tag{5.4}$$

Proof: Let the length of R_A and R_B be t_a and t_b, respectively. We define the ordered pair (t_a, t_b), where $(t_a, t_b) < (t'_a, t'_b)$ if and only if $t_a < t'_a$, or $t_a = t'_a$ and $t_b < t'_b$. We now prove the theorem by induction on the ordered pair (t_a, t_b).

(1) As the initial condition, we prove that the theorem is true for $(t_a, t_b) = (0, 0)$. With no Syncs in either input, the merge algorithm simply forms a Cartesian Product over the two input streams and the result has the form

$$D^c = \Sigma_\emptyset D^c = D^a \times D^b = \Sigma_\emptyset D^a \times \Sigma_\emptyset D^b$$

which satisfies (5.4) with $v_a = v_b = v_c = \emptyset$.

(2) In general, we prove that the theorem holds for $(t_a, t_b) = (t'_a, t'_b)$ provided that it holds for all $(t_a, t_b) < (t'_a, t'_b)$, where t'_a and t'_b are two arbitrary non-negative integers.

To merge two input streams $\Sigma_{R_A}(D^a)$ and $\Sigma_{R_B}(D^b)$ with $|R_a| = t'_a$ and $|R_b| = t'_b$ respectively, $merge(\Sigma_{R_A}(D^a), \Sigma_{R_B}(D^b))$ is called. When comparing the first elements in two input streams, one of the following three conditions hold. (To be consistent with the program in Figure 5.4, let A and B be the first elements of $\Sigma_{R_A}(D^a)$ and $\Sigma_{R_B}(D^b)$, respectively.)

(a) If A and B are both Sync signals and A precedes B, or A is a Sync signal and B is not, then the second condition in procedure *merge* is true and the first input stream is divided into two partial streams by *scanto*. The first partial stream is the input data enclosed by the first pair of Sync signals A, which has the form $\Sigma_{R_A-\{i\}}(D_1^{a'})$, where $A = S_i$ and $D_k^{a'} = \{D_{v_a}^a | v_a[1] = k\}$. The second partial stream has the same syntactic structure as the original stream except that the number of Sync signals A is one less than the original stream. After A is put in the output queue, two merges are called consecutively to merge the two parts of the first stream with the second stream. Since $|R_A - \{i\}| = t'_a - 1$ and $(t'_a - 1, t'_b) < (t'_a, t'_b)$, we can derive the result of $merge(\Sigma_{R_A-\{i\}}(D_1^{a'}), \Sigma_{R_B}(D^b))$ by induction. The second merge call is the same as the original call and the operation described above is repeated until the first input stream is exhausted. When the merge process terminates, a sequence of outputs

$$(S_i, \Sigma_{R_{C'}}(D_1^{c'}), S_i, \Sigma_{R_{C'}}(D_2^{c'}), S_i, \ldots, \Sigma_{R_{C'}}(D_n^{c'}))$$

is generated, where $R_{C'} = (R_A - \{i\}) \sqcup R_B$, and $D_{k,v_{c'}}^{c'} = D_{k,v_{a'}}^{a'} \times D_{v_b}^b$ with $v_{c'} = v_{a'} \sqcup v_b$ and $v_{a'}[i] = v_a[i+1]$ for all $i : 1 \le i \le (t'_a - 1)$ and $k \le n$. Therefore, the output sequence can be represented as $\Sigma_{\{i\} \sqcup R_{C'}}(D^c)$, where $D_{v_c}^c = D_{v_a}^a \times D_{v_b}^b$ with $v_c = v_a \sqcup v_b$, and $D_{v_c}^c = D_{v_c[1],v_{c'}}^{c'}$ with $v_{c'}[i] = v_c[i+1], 1 \le i < |R_A \cup R_B|$. Thus, the theorem is true for (t'_a, t'_b) under this condition.

(b) If A and B are both Sync signals and B precedes A, or B is a Sync signal and A is not, then the third condition is true and the second input stream is partitioned into two partial streams. The proof is the same as case (a) except that the two input streams are switched. The proof is established on the assumption that $merge(\Sigma_{R_A}(D^a), \Sigma_{R_B-\{i\}}(D_k^{b'}))$ returns the correct answer because $(t'_a, t'_b - 1) < (t'_a, t'_b)$.

(c) If A and B are the same Sync signal, say A=B=S_i, the last condition of the merge program is true. Both the input streams are partitioned into two partial streams by *scanto* and merge is applied on the corresponding parts of the two input streams. The output of the first call $merge(\Sigma_{R_A-\{i\}}(D_1^{a'}), \Sigma_{R_B-\{i\}}(D_1^{b'}))$ can be derived from this theorem because $(t'_a - 1, t'_b - 1) < (t'_a, t'_b)$ and by assumption the theorem holds. The second merge is the same as the original call except that the first S_i

and the following data segments before the next occurrence of S_i are removed from the original streams. Such a merge is called recursively until both input streams are exhausted. When the merge process terminates, we get a sequence of outputs

$$(S_i, \Sigma_{R_{C'}}(D_1^{c'}), S_i, \Sigma_{R_{C'}}(D_2^{c'}), S_i, \ldots, \Sigma_{R_{C'}}(D_n^{c'}))$$

where $R_{C'} = (R_A - \{i\}) \sqcup (R_B - \{i\})$, and $D^{c'}_{k,v_{c'}} = D^{a'}_{k,v_{a'}} \times D^{b'}_{k,v_{b'}}$, with $v_{c'} = v_{a'} \sqcup v_{b'}$, $v_{a'}[i] = v_a[i+1]$, $\forall i : 1 \leq i \leq (t'_a - 1)$, $v_{b'}[i] = v_b[i+1]$, $\forall i : 1 \leq i \leq (t'_b - 1)$, and $k \leq n$. Therefore, the output sequence can be represented as $\Sigma_{\{i\} \sqcup R_{C'}}(D^c)$ and $D^c_{v_c} = D^a_{v_a} \times D^b_{v_b}$ with $v_c = v_a \sqcup v_b$ where $D^c_{v_c} = D^{c'}_{v_c[1], v_{c'}}$ and $v_{c'}[i] = v_c[i+1], 1 \leq i < |R_A \cup R_B|$. Thus, the theorem is true for (t'_a, t'_b) under this condition.

With the above three conditions, we have proved the theorem for $(t_a, t_b) = (t'_a, t'_b)$ provided that the theorem holds for $(t_a, t_b) < (t'_a, t'_b)$. In addition, the theorem holds for $(t_a, t_b) = (0, 0)$. By induction, the theorem is true for any pair of $(t_a, t_b) \geq (0, 0)$. In other words, the theorem is true for any two arbitrary input streams. ∎

From Theorem 5.2 we can derive the merge result with two arbitrary input streams. The remaining task is to show that the merge result is correct. Given a process with two inputs, a *legal input combination* is an input pair such that the elements of the input pair are derived from the same output of a common ancestor along the two input paths of the process. For instance, a simple data flow graph with four processes is shown in Figure 5.7, in which process a is a common ancestor along the two input paths of d. We should combine only the inputs of process d that are derived by the same output of process a. Notice that such a common ancestor is marked as a Sync generator in our Model. In other words, the intersection of the Sync sequences of the two input streams consists of all the common ancestors in the input paths of the process.

Let D_{v_R} be a data segment in an input stream $\Sigma_R(D)$. The rank vector v_R indicates the ranks of all the Sync signals in R from which D_{v_R} is derived. Therefore, the inputs that are derived from the k-th output of the Sync generator $R[i]$ are the union of all the data segments, D_{v_R}, such that $v_R[i] = k$.

Theorem 5.3. *The merge algorithm generates all the legal input combinations to a pro-*

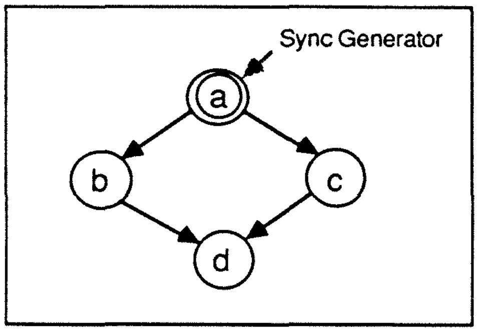

Figure 5.7. A Data Flow Graph with Multiple Paths

cess.

Proof: Suppose there are k common Syncs in the two input streams; i.e., $R_A \cap R_B = \{A_i | 1 \leq i \leq k\}$. Define a predicate $P(v_R)$ such that

$$P(v_R) \equiv \bigwedge_{1 \leq i \leq k} (v_R.A_i = a_i).$$

$P(v_R)$ is true if the ranks of the common Syncs $A_1, \ldots, A_k$ are $a_1, \ldots, a_k$, respectively. We have to prove that the merge result that corresponds to the a_i-th output of the Sync generator A_i, $1 \leq i \leq k$, is the Cartesian Product of the portions of the two input streams that are derived from the a_i-th output of the Sync generator A_i, $1 \leq i \leq k$. The above relation can be formulated as follows:

$$\bigcup_{\forall v_c : P(v_c)} D^c_{v_c} = \bigcup_{\forall v_a : P(v_a)} D^a_{v_a} \times \bigcup_{\forall v_b : P(v_b)} D^b_{v_b}. \tag{5.5}$$

(5.5) can easily be derived from (5.4). First add a big union $\bigcup_{\forall v_c : P(v_c)}$ to both sides of (5.4). Then separate the unions at the right-hand side into two independent sets of unions, move the unions inside the CP, and associate the first set of unions to $D^a_{v_a}$ and the second set of unions to $D^b_{v_b}$.

$$\begin{aligned}
\bigcup_{\forall v_c : P(v_c)} D^c_{v_c} &= \bigcup_{\forall v_c : P(v_c)} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\forall (v_a \sqcup v_b) : P(v_a) \wedge P(v_b)} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\forall v_a : P(v_a)} \bigcup_{\forall v_b : P(v_b)} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\forall v_a : P(v_a)} D^a_{v_a} \times \bigcup_{\forall v_b : P(v_b)} D^b_{v_b}
\end{aligned}$$

Therefore, we can conclude that the merge algorithm gives all the legal input combinations. ∎

With the above theorems, we would like to show that the Sync Model is sound and complete. *Soundness* means that every answer generated by the Sync Model is a correct answer and *completeness* means that the Sync Model generates all the correct answers. In order to prove soundness and completeness of our model, we shall first find the relationship between the search tree of a given program and its process tree generated by our model.

The process tree of an original goal represents the complete search space of the goal. In the process tree, there exists a unique subtree corresponding to every successful computation. Let $G_0, G_1, \ldots, G_n$ be a successful computation; then the corresponding subtree, T, can be derived through a set of subtrees $T_0, T_1, \ldots, T_n$, such that $T_n = T$ and T_i corresponds to a partial computation $G_0, G_1, \ldots, G_i$ and $T_i \subset T_{i+1}$:

1. G_0 is mapped onto the root of the process tree.

2. T_{i+1} is derived from T_i as follows:
   (a) $T_i \subset T_{i+1}$,
   (b) If G_{i+1} is derived from $G_i = A_1, \ldots, A_k, \ldots, A_n$ with a selected goal A_k and a selected clause $C_{kj} : (A : -B_1, \ldots, B_m)$ (the j-th clause in the definition whose head matches A_k), then the j-th descendant (an OR process holding C_{kj}) of the k-th leaf node (an AND process holding A_k) of T_i is added into T_{i+1}, as well as all the descendants (AND processes containing B_1, B_2, to B_m, respectively) of the newly added node.
   (c) If C_{kj} is a unit clause, then A_k is reduced to empty. The j-th descendant of the k-th leaf node of T_i is added to T_{i+1}. The newly added node is a leaf node of the process tree that terminates successfully.

Theorem 5.4. *The Sync Model is sound and complete.*

Proof: **Soundness:** We first prove that any solution generated by the Sync Model is a correct answer. We prove the soundness in two steps: (1) to prove that any minimal subtree in the Sync Model that produces an answer corresponds to a successful computation, and

(2) to prove that such a subtree generates the same answer as the corresponding successful computation.

(1). A minimal subtree is a subtree that contains no failure nodes. It is obvious that a minimal subtree that produces an answer is similar to the subtree defined for a successful computation, i.e., it starts from the root, expands by including exactly one descendant OR process for each of its AND processes and all the descendant AND processes for each of its OR processes, and ends with leaf nodes which terminate successfully. Furthermore, we can find a computation $G_0, G_1, \ldots, G_n$ corresponding to such a subtree†. The only thing left unproven is whether or not this computation is a successful computation; i.e., $G_n = \emptyset$. A leaf node of this kind of subtree is an OR process which contains a unit clause, say C_j, and succeeds in unification with the goal in its father AND process, say A_k. Therefore, if a goal statement G_j contains A_k, A_k will be reduced to empty with C_j in another G_l and $j < l \leq n$. Since all such A_k will be reduced to empty eventually, G_n will become empty for a finite n. Thus, any minimal subtree that produces an answer corresponds to a successful computation.

(2). By induction. First choose an OR process in a subtree that corresponds to a successful computation. Assume that this OR process contains a goal g and a clause "$g1 : -p_1, p_2, \ldots, p_n$." Let $X_1, X_2, \ldots, X_m$ be the variables of this clause. The successful computation gives a unique solution to these variables, say $t_1, t_2, \ldots, t_m$. Moreover, let each p_i contain a set of input variables and a set of output variables. The input-variable set and the output-variable set of any p_i are disjoint and both of them are subsets of $(X_1, \ldots, X_m)$.

Figure 5.8 shows a part of this subtree, where some communication channels exist among the AND processes p_1 to p_n. Assume that the subtree under each p_i produces the correct solutions for the output variables of p_i if the input variables are bound to the correct values. Here, the correct solution of a variable X_i is meant to be t_i. Therefore, any process p_i that has no input variable will generate the correct solutions to its output variables. Furthermore, any p_i with a nonempty input-variable set will produce the correct solutions

† Different selection functions may generate different computations corresponding to the same subtree. Nevertheless, all those computations generate the same answer.

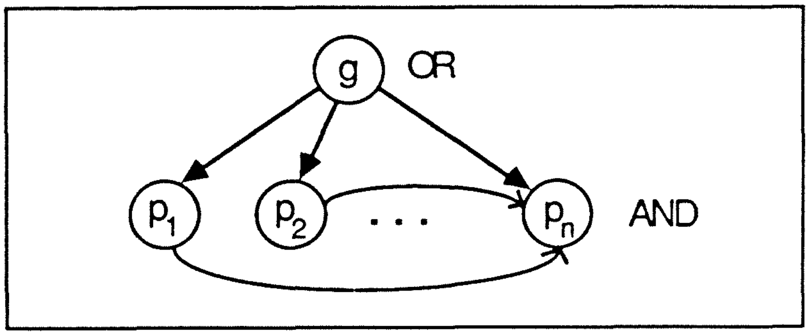

Figure 5.8. A Subtree for $g1 : -p_1, p_2, \ldots, p_n$.

to its output variables if the producers of its input variables generate the correct solutions. The above statement is obviously true if p_i has only one input variable. It is also true if p_i has more than one input variable because the merge algorithm in p_i always generates the correct input combinations by Theorem 5.3. Therefore, the OR process generates the correct solution for its goal g assuming the subtrees under each p_i are correct. Furthermore, if the OR process we chose contains a unit clause, it must be a leaf node and it generates the correct solutions to the output variables of its goal. Thus, by induction, the subtree corresponding to a successful computation will generate the correct solution for that computation.

From (1) and (2), we have proved that the Sync Model is sound.

Completeness: Since any successful computation can be mapped onto a subtree in the Sync Model and each subtree generates the correct solution for the corresponding computation from the above proof, we conclude that the Sync Model generates all the solutions for a given program and therefore it is complete. ∎

5.5 Summary

The merge algorithm is performed in both the AND processes and the OR processes. When an AND process has more than one input channel, the merge algorithm is applied to merge the input streams from all the channels to form the legal input combinations. When an OR process has more than one input channel from its descendants, a similar merge algorithm is applied to merge the partial solution streams to form the correct answers of this OR process.

The basic operation of the merge algorithm is the Cartesian Product of the input streams. An efficient implementation of Cartesian Product, which repeatedly forms the Cartesian Products over the available portions of the input streams, is designed so that the process needs not wait for the inputs from a slow channel. With the appearance of the Sync signals, an input stream is partitioned into several portions separated by the Sync signals and the Cartesian Products are formed over portions of the input streams instead of the entire streams. The general merge algorithm generates the Cartesian Products over the portions of the input streams that originate in the same output of the common ancestors along the input paths of this process.

The merge algorithm combined with Synchronization signals makes the data-driven model possible. It solves a difficult problem common to all the demand-driven models, i.e., how and where to backtrack the computation when a process fails. The major distinction of our Model from previous models is that the processes in our model generate and send out all the solutions without explicit requests. Each process that successfully generates all its solutions can terminate right after the solutions are sent out. The advantages are twofold. First, we can avoid the complex control in backward execution [11], which is used to simulate backtracking. Second, a process can release itself as soon as it has generated all the solutions. Processors can be utilized more efficiently without many suspended processes.

The proper functioning of the merge algorithm assures the correctness of our Sync Model. A proof of the correctness of the merge algorithm has also been presented in this chapter. Based on the correctness of the merge algorithm, we then proved that our Sync Model is sound and complete.

Chapter 6

Improvement of The Sync Model

In this chapter, the Sync Model described in Chapter 3 is first extended to allow partially instantiated terms in variable bindings. Dynamic links are added to the data flow graph during the computation to enforce the "one producer per variable" rule for the newly generated variables. Afterwards, the Sync Model is further improved to handle stream parallelism, tail recursion optimization, and deterministic programs more efficiently.

6.1 Partially Instantiated Terms

6.1.1 Dynamic Links

As we mentioned in Chapter 3, the basic Sync Model is restricted such that all output variables generated by a process must be fully instantiated. Although it rarely happens, binding a variable to a partially instantiated term is possible in CLP. When the producer of a shared variable binds the variable to a partially instantiated term, a binding conflict may occur if this shared variable has more than one consumer. For instance, consider the following clause:

$$a :- b(X), c(X), d(X). \tag{6.1}$$

All the AND processes in the clause body share a common variable X. The ordering algorithm selects the leftmost AND process b as the producer of X, and c and d as the consumers of X. The data flow graph is shown in Figure 6.1.a. Notice that processes c and d can be executed concurrently. If the process *b* binds X to another variable, say $f(Y)$; then variable Y will be produced by both processes *c* an *d*, and a binding conflict may occur. In this situation, one of the two processes should be selected to be the producer of Y and a new channel for Y should be added from its producer to its consumer. Let's select process *c* to be the producer of Y, then a new link directed from *c* to *d* should be added to the data flow graph and the new graph is shown in Figure 6.1.b. Such a link is called a *dynamic link* since it exists only when X is bound to another variable and X is sent to more than one consumer.

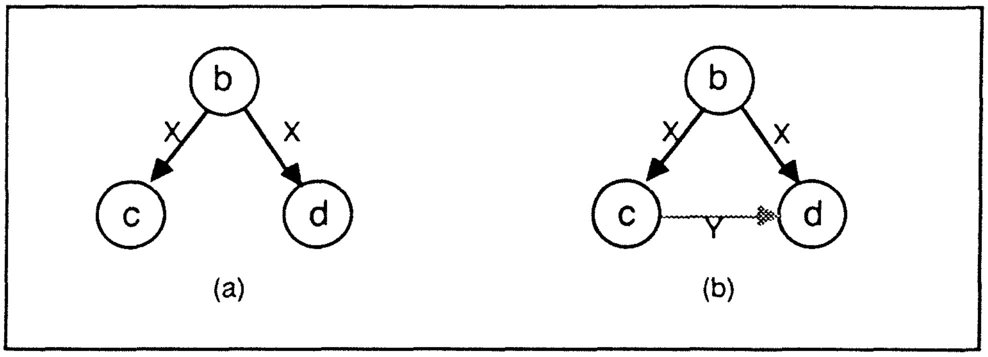

Figure 6.1. Data Flow Graph for Clause (6.1)

Another situation that may cause binding conflicts is when the bindings of the two different shared variables are dependent on each other. For instance, the clause (6.2) has two shared variables X1 and X2, and each variable is consumed by one process.

$$a :- b(X1, X2), c(X1), d(X2). \tag{6.2}$$

The ordering algorithm constructs the data flow graph for the three AND processes as shown in Figure 6.2.a, where processes *c* and *d* can be executed concurrently. If process *b* binds both X1 and X2 to a partially instantiated term containing the same variable, say Y (e.g. X1 is bound to $f(Y)$ and X2 to $g(Y)$), then a binding conflict of Y may occur. Likewise, one of the two processes, say *c*, is selected to be the producer of Y and a dynamic link directed from *c* to *d* is added to transmit the value of Y (Figure 6.2.b).

A more complex example is given in Figure 6.3. Five AND processes are interconnected through the communication channels marked by the shared variables transmitted along those channels (Figure 6.3.a). There are three consumers *b*, *c* and *d* for variable X. When process *a* binds X to a partially instantiated term, say f(Y), one of the consumers of X is

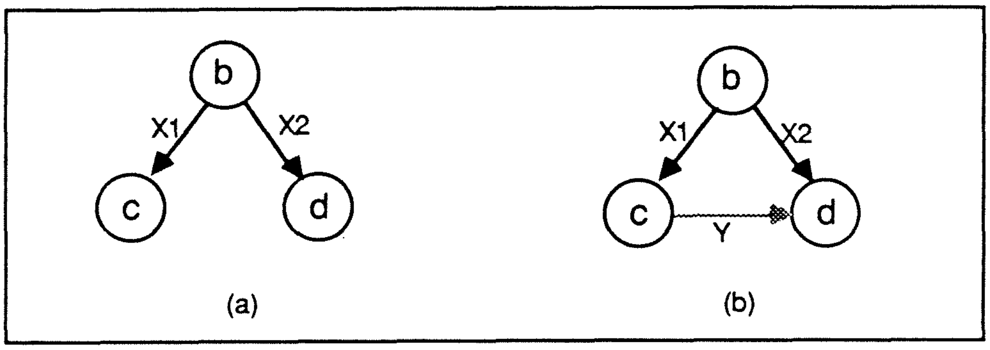

Figure 6.2. Data Flow Graph for Clause (6.2)

selected as the new producer of Y, say *b*, and dynamic links (*b*, *c*) and (*b*, *d*) are added to the graph (Figure 6.3.b). Furthermore, if process *b* binds its two output variables, Y and U, to $g(Z)$ and $h(Z)$, respectively, then one of the three consumers of Y and Z (i.e., *c*, *d*, and *e*) should be selected to be the producer of Z, say *c*, and dynamic links (*c*, *d*) and (*c*, *e*) should be added to the graph for the new variable Z (Figure 6.3.b). At last, we add dynamic links (*d*, *e*) and (*d*, *f*) in case process *c* binds both Z and V to another variable T. The links (*d*, *e*) and (*d*, *f*) are needed because process *d* and *e* are the consumers of Z, process *f* is the consumer of V, and process *d* is selected to be the producer of T. This dynamic link information is made available after the variable bindings are produced.

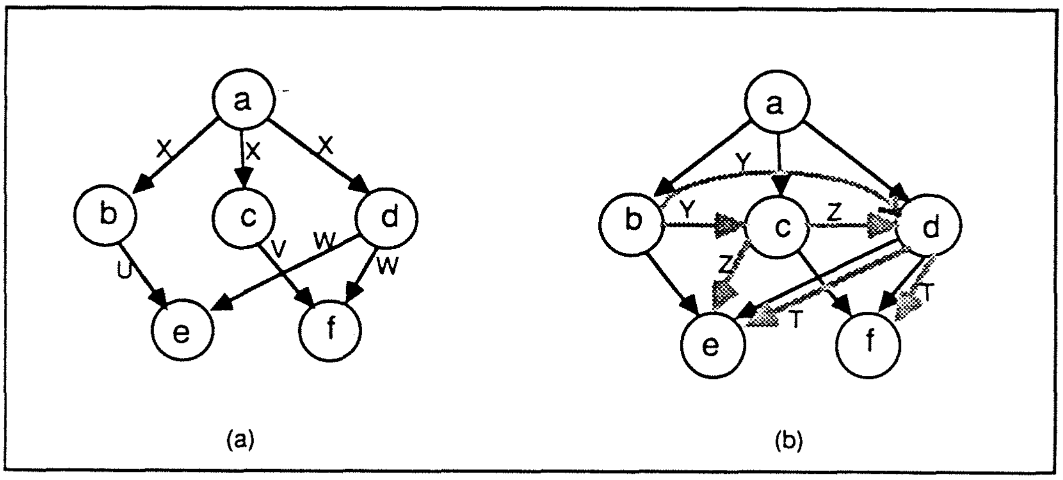

Figure 6.3. Another Example of Dynamic Links

In summary, the dynamic links are needed when an AND process binds one or more output variables to another variable and there is more than one consumer for those variables. One of those consumers is selected as the producer of the new variable and the dynamic links are directed from the new producer to all the other consumers. The information about dynamic links is not provided during the construction of the data flow graph. Instead, such information is generated and sent to the selected producer of the new variable when an AND process binds some output variables to partially instantiated terms. A simple test on the binding values of all the output variables is sufficient to determine whether dynamic links are needed and how they are directed. Such a test is similar to DeGroot's type checking [14], except that we do the same check in every AND process without consulting the complex graph expression proposed by DeGroot. Our dynamic link construction mechanism is more efficient than Conery's method [11], in which an ordering algorithm is called every time a new variable binding is generated.

6.1.2 Multiple Paths and Dynamic Links

The detection of multiple paths is done when the data flow graph is constructed. The construction of dynamic links may cause more multiple paths so that more processes should be marked as Sync generators. When a set of dynamic links is added to the data flow graph, the source node of these dynamic links (i.e., the producer of the new variable) becomes a new Sync generator if the node that binds a variable to the partially instantiated term is also a Sync generator. For instance, in Figure 6.4 a multiple path exists between a and d. When process a binds X to another variable Y, a dynamic link (b, c) needs to be added into the graph. The insertion of dynamic link (b, c) makes process b a Sync generator because a is a Sync generator and a multiple path between b and d is detected. Let process a bind X to x_1 and $f(Y)$; process b bind Z to z_1 with input x_1 and bind (Y,Z) to (y_1, z_2), (y_2, z_3) with input $f(Y)$. The input combination at process c is $x_1, f(y_1), f(y_2)$, and the output stream corresponding to each input is w_1, w_2, w_3. With the appearance of the Sync signals from processes a and b, the two input streams received at process d should be $(S_a, z_1, S_a, S_b, z_2, S_b, z_3,END)$ and $(S_a, w_1, S_a, S_b, w_2, S_b, w_3,END)$, respectively. The input combination at process d becomes $(S_a, (z_1, w_1), S_a, S_b, (z_2, w_2), S_b, (z_3, w_3),END)$. It is clear that without the Sync signals S_b, we can't derive the above input combination and therefore the proper synchronization can't be achieved.

Is there any other process which may become a Sync generator when a set of dynamic links is added to a process? Intuitively, the process that binds the variable to another variable would possibly be a Sync generator. For example, process b in Figure 6.1.b is

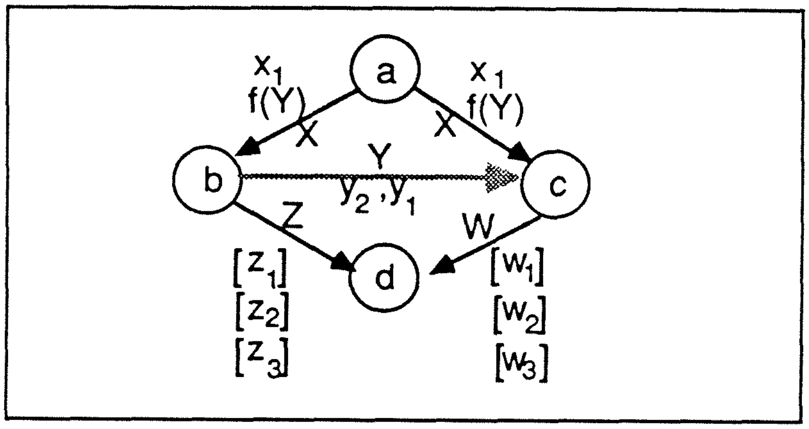

Figure 6.4. Extra Sync Generator Caused by Dynamic Links

such a process. The insertion of dynamic links indeed creates a multiple path between this kind of process and all its consumers except the one which is selected to be the producer of the new variable, e.g., a multiple path between b and d in Figure 6.1.b. Nevertheless, that process needs not be a Sync generator because the dynamic links exist temporarily to transmit the bindings for only one output variable. If the process binds an output variable to several partially instantiated terms, the new variables in these terms are distinguished by different names. Therefore, there is no synchronization problem in those multiple paths. Consider the data flow graph in Figure 6.1. Let process b bind X to $(x_1, f(Y), g(Z))$. Notice that Y and Z always have different names. Assume that process c binds Y to (y_1, y_2) and Z to (z_1, z_2); then the input stream along link (b, d) is $(x_1, f(Y), g(Z)$,END) and the input stream along (c, d) is $(y_1, y_2$,END,z_1, z_2,END), and the input combination of process d is $(x_1, f(y_1), f(y_2), g(z_1), g(z_2)$,END). Since an END signal is appended to the binding values of a new variable along a dynamic link, it is not necessary to add more Sync signals to do the synchronization.

6.1.3 The Extended Sync Model

In order to transmit the information of dynamic links between sibling AND processes, a new type of message, the *channel* message is introduced. The *channel* message contains channel information for dynamic links. It is generated by the process that produces partially instantiated outputs, and is sent to the process that is selected to be the producer of the newly generated variable.

In the extended Sync Model, the program of the OR process is the same as described in Section 3.3. The program for the AND process is modified so that the AND process can recognize partially instantiated terms in the variable bindings and can handle *channel* messages.

In routine AND_PROC (Figure 3.5), if an input contains another new variable, the dynamic links for the new variable should have been received by MERGE_AND and appended to its output channel list. If the producer of the input variable that is bound to another variable is a Sync generator and the process itself is not already a Sync generator, the AND_PROC then temporarily marks itself as a Sync generator.

When the process receives an answer from one of its descendants, it has to check if there is any partially instantiated term in the answer as well. The procedure *output* in Figure 3.5 is rewritten as in Figure 6.5. It first checks to see if there is any new variable in the answer or if the bindings for any two different variables depend on each other; i.e., both variables are bound to a third variable. If so, the new variable name is changed if necessary to make it distinguishable from other local variables' names. Then, one consumer of the partially instantiated variables is chosen to be the producer of the newly generated variable. The one we chose is the process with the smallest identifier to assure the acyclicity of the data flow graph with dynamic links added. Then, a set of dynamic links directed from the newly selected producer to the rest of the consumers of the partially instantiated variables is created, and the information of these links is sent to the new producer via a *channel* message. Lastly, the answer is sent to its consumers according to the channel list.

```
procedure output(answer);

[sync ∨ needSync → send_to_all_channels(binding(SYNC_id))
|otherwise → skip
];
[var_in(answer) → X1:='the variable that is bound to a partially instantiated term';
                  Y1:='the variable contained in the partially instantiated term';
                  C:='select a producer for Y1';
                  D:='dynamic links for Y1';
                  send(C,channel(D));
                  send_each_channel(binding(answer))
|otherwise → send_each_channels(binding(answer))
].
```

Figure 6.5. Procedure *output*, Which Handles Partially Instantiated Terms

The routine MERGE_AND has to handle two types of input messages, *channel* and *binding*. The program for MERGE_AND is shown in Figure 6.6. When a *channel* message is received at an input channel, the channel information for the dynamic links carried in the message is appended to the output channel list of the AND process. When a *binding* message is received, the routine checks to see if there are any new variables in the input. If a variable is found in the input and the variable is not an output variable (i.e., it is not in the output channel list), the routine constructs a new input channel for this variable and waits to receive messages from the channel. Upon receiving one variable binding from the new channel, MERGE_AND substitutes the binding value for the variable in the previous message and then puts the message with substituted value into the input buffer. The above substitution process is repeated until all the messages on the new input channel have been received and an END signal is detected. If the input contains an output variable or no variables at all, MERGE_AND simply puts the message into the input buffer.

```
 MERGE-AND(input-list);

% When the next buffer entry is empty, read in all the messages arrived at the input channel
[empty(buf[i,j]) → m:=j;
        *[¬ empty(IN[i]) →
                      [IN[i]?channel(input) → append_channel_list(input)
                      |IN[i]?binding(input) →
                              [var_in(input) → X:=new_var(input);
                                    [in_channel_list(X) → % X is an output variable
                                                           buf[i,m]:=input;
                                                            m:=m+1
                                    |otherwise → *[IN[K+1]?binding(END) → skip
                                                     % IN[K+1] is the channel for X
                                                    |IN[K+1]?binding(new_var) →
                                                         new_in:=replace(input,new_var);
                                                         buf[i,m]:=new_in;
                                                         m:=m+1
                                                   ]
                                    ]
                              |otherwise → buf[i,m]:= input;
                                           m:=m+1
                              ]
                      ]
        ]
|otherwise → skip
];
```

Figure 6.6. Program for MERGE_AND

6.2 Stream Parallelism

Stream parallelism is another source of parallelism inherent in logic programming [10]. It is the pipelining of list data structures among AND processes. If a list is shared by two AND processes, the list elements can be transmitted from one process to the other process one at a time without waiting until all the list elements are produced. This allows the two processes to run in parallel. This kind of parallelism is not considered in the basic Sync Model but can be implemented in the Sync Model with minor modifications.

A *stream variable* is defined as a list of elements of any type; the elements in the list are produced one at a time by a process and consumed one at a time by another process. An AND process containing a goal which is defined tail recursively is usually a producer or a consumer of some stream variables. Let a tail recursive clause be as follows:

$$p([X|L], \ldots) :- \ \ldots, p(L, \ldots). \tag{6.3}$$

Then the list variable L is a candidate for a stream variable. The values of a stream variable are transmitted not only horizontally between sibling AND processes but also vertically between an AND process and an OR process. For instance, if L is a stream variable in clause (6.3), then the values of L will be transmitted between the OR process containing that clause and the descendant AND process that executes the literal $p(L, \ldots)$ in the clause body. The values of L may be sent upwards or downwards depending on whether L is an output variable or an input variable in $p(L, \ldots)$.

It is important to distinguish a stream variable from a partially instantiated term. Assuming that an AND process, p, receives an input binding which happens to be a list, say $[x|L]$, with L unbound. If L is a stream variable, then L becomes an input variable of p. The value of L will be received later from the same input channel and propagated to all the active OR descendants of p. Otherwise, L is a partially instantiated variable. Therefore, it is necessary to determine which variables are stream variables in the clause body.

A shared variable, L, in a clause body is a stream variable if the following conditions hold:

1. For all the literals containing L, the argument corresponding to L in the definition of the literal is a list.

2. At least one definition clause of each literal containing L is recursively defined in the list argument corresponding to L.

A clause is called ***recursively defined in a list variable*** if (1) it is tail recursive in this variable, or (2) if the list is an argument of a literal in the clause body, and that literal is recursively defined in this argument.

The detection of stream variables can be done statically before the execution of the program. After the data flow graph is constructed in an OR process, any shared variable that is a stream variable will be annotated with a different set of annotations: "!!" for output stream variable and "??" for input stream variable to make them distinguishable from non-stream variables.

The stream variable annotations can be added optionally into the program by the programmer as well. When added by the programmer, the above definition for the stream variables needs not to be satisfied.

For instance, Figure 6.7 is the quicksort program mentioned in Chapter 4. The clauses are numbered from (1) to (7). Clause (7) for *append* is tail recursive in the first and the third arguments. Clause (3) for *split* is tail recursive in the second and the third arguments. Clause (4) for *split* is tail recursive in the second and the fourth arguments. Clause (1) for *sort* is also recursively defined because the first argument of the clause head is consumed by literal *split* and the definition of literal *split*, i.e., clauses (3) and (4) (except clause (5) which is for the boundary condition), are recursively defined in the second argument. Therefore, all three predicates: *sort*, *split* and *append* are recursively defined.

Clause (1) has four shared variables in its body: *Smaller*, *Larger*, *Sorted*1, and *Sorted*2. *Smaller* is a stream variable because (a) the third argument in the definition of *split* (from clause (3)) and the first argument of *sort* are both lists, (b) *split* is recursively defined in *Smaller* from clause (3), so is *sort*. Likewise, *Larger* and *Sorted*1 are stream variables as well. But *Sorted*2 is not a stream variable because *append* is not recursively defined in this variable. If $sort([2,1,3],L)$ is called, the data flow graph of clause (1) is constructed by the ordering algorithm and the clause with the annotated argument is shown in Figure 6.8.

Replacing the annotations of *Smaller*, *Larger*, and *Sorted*1 by the stream annotations,

```
(1)    sort([X|Unsorted],Sorted) :- split(X,Unsorted?,Smaller,Larger),
                                     sort(Smaller?,Sorted1),
                                      sort(Larger?,Sorted2),
                                      append(Sorted1?,[X|Sorted2],Sorted).
(2)    sort([],[]).

(3)    split(X,[A|L],[A|Smaller],Larger) :- A<X → split(X,L,Smaller,Larger).
(4)    split(X,[A|L],Smaller,[A|Larger]) :- A≥X → split(X,L,Smaller,Larger).
(5)    split(X,[],[],[]).

(6)    append([],L,L).
(7)    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

:- sort([2,1,3],L).
```

Figure 6.7. Quicksort Program

```
sort([2,1,3],L) :- split(2,[1,3],Smaller!,Larger!),          (1)
                     sort(Smaller?,Sorted1!),                   (2)
                      sort(Larger?,Sorted2!),                   (3)
                      append(Sorted1?,[2|Sorted2?],L!).         (4)
```

Figure 6.8. The Clause after the Ordering Algorithm Has Been Applied

we got the new clause in Figure 6.9.

```
sort([2,1,3],L) :- split(2,[1,3],Smaller!!,Larger!!),        (1)
                     sort(Smaller??,Sorted1!!),                 (2)
                      sort(Larger??,Sorted2!),                  (3)
                      append(Sorted1??,[2|Sorted2?],L!).        (4)
```

Figure 6.9. The Clause with Stream Variables

To realize stream parallelism, once an element of a stream variable is generated by an OR process, the value is to be sent out without waiting until all the descendants of the OR process terminate. The basic Sync model has to be modified as follows to implement the stream parallelism.

1. An OR process with an non-unit clause replaces the annotations of stream variables by stream annotations after the ordering algorithm is applied.

2. The unification rule is extended to handle stream variables as follows:
   A stream variable, X, can be unified with a list structure, a variable or another stream variable.

a. To unify a stream variable X in the goal with a list structure $[x|L]$ in the clause head results in marking L as a stream variable in the body, and instantiating X to $[x|L]$.

b. To unify X in the goal with a variable Y in the body makes Y instantiated to X.

c. To unify X with another stream variable Y makes Y instantiated to X if Y has the same annotation as X. Otherwise, the unification fails.

3. If an OR process contains an output stream variable, say X, the binding value of X is ready to be sent back once the unification succeeds and all the guard processes terminate successfully. If X is remained uninstantiated after unification, the process will wait until at least the head of X is bound to an atomic term, and then send the result back. If the OR process contains some other output variables, the values of X will be merged with the values of other outputs first so that multiple solutions produced by this process can be synchronized.

4. A binding of a stream variable X is of the form $\langle X, [x|L]\rangle$, where x is the head of the list and L is a variable. As usual, the stream of X is a sequence of bindings, Sync signals and an "END" signal at the end. The merge algorithm, which either merges the values of the input variables in an AND process or merges the values of the output variables in an OR process, works the same for a stream variable. Notice that the merge algorithm only merges the input stream of the original stream variable appearing in the goal, i.e., X. The values of L, which again is a stream variable, will be received and sent out directly, so is the tail of L, etc.. An AND process propagates the values of the stream variables to all its descendants and an OR process sends the values of stream variables to its father process.

5. If an OR process contains an input stream variable or an AND process contains an output stream variable, say X, the values of X and all the stream

variables created for the tail of X will be received and sent out immediately after being received. In an OR process, these values are received from its father process and sent out to the descendants that contain X in their goals. In an AND process, these values are received from its descendants and sent out through the output channels of X.

6. Each process, p, maintains a ***tail list*** for each of the output stream variables in its goal. Originally, the tail list consists of the original output stream variable, X. Once a binding $\langle X, [x|L] \rangle$ is received, a new stream variable L is added to the list. The new variable is renamed, if necessary, to keep it distinguishable from the other variables in the list. When an "END" signal is received for a variable in the list, this variable is removed from the list. The descendant process that is the sender of a stream variable, X, will be removed from the active descendant list of p when the tail list of X is reduced to empty. Process p terminates when its active descendant list becomes empty.

With stream parallelism, it is possible to handle an unbounded list. Pipelining of adjacent processes in the data flow graph speed up the computation, especially when a long list is shared by these adjacent processes.

It is obvious that the major change of our Sync Model in order to realize stream parallelism is the different communication protocols for stream variables. In addition, the unification rules are changed to handle the stream variables, and the binding of a stream variable is sent out when an OR process reaches the commit operator, i.e., when all the guard literals have been evaluated to true and the AND literals are not yet executed. Therefore, the language semantics is changed by the variable annotations and the commit operator. With a shared stream variable between two AND literals, we may need to form a long communication path for this stream variable, directed from the bottom of one branch of the process tree up to the producer of this variable, then routed to the other AND process and directed down to the bottom of the process tree. Such a long communication path results in high overhead. Tail recursion optimization discussed in the next subsection will compensate this drawback.

In order to synchronize the multiple solutions in our Model, we sacrifice the stream parallelism a little bit in step (3) by waiting for the values for the other outputs. Full stream parallelism can be achieved in the Deterministic Programming mode which will be described in Section 6.4.

6.3 Tail Recursion Optimization

Tail recursion is one common way in logic programming to realize iterations. A clause with the form

$$p([X|L]) :- use(X), p(L).$$

is a typical clause with tail recursion. If there is a unit clause "p[ ]." in the program that defines the termination condition for p, then in the process tree, such a clause forms a long chain as shown in Figure 6.10. In order to complete the computation, the input has to pass down to the bottom of the chain, and the result will be popped all the way back. There is a great communication overhead in this implementation. To reduce the overhead, we can form a loop and reuse the topmost process as shown in Figure 6.11. With this technique, the communication overhead is greatly reduced and processes are more efficiently utilized. But the disadvantage of this scheme is that the computation is serialized so that potential parallelism in the clause may be lost.

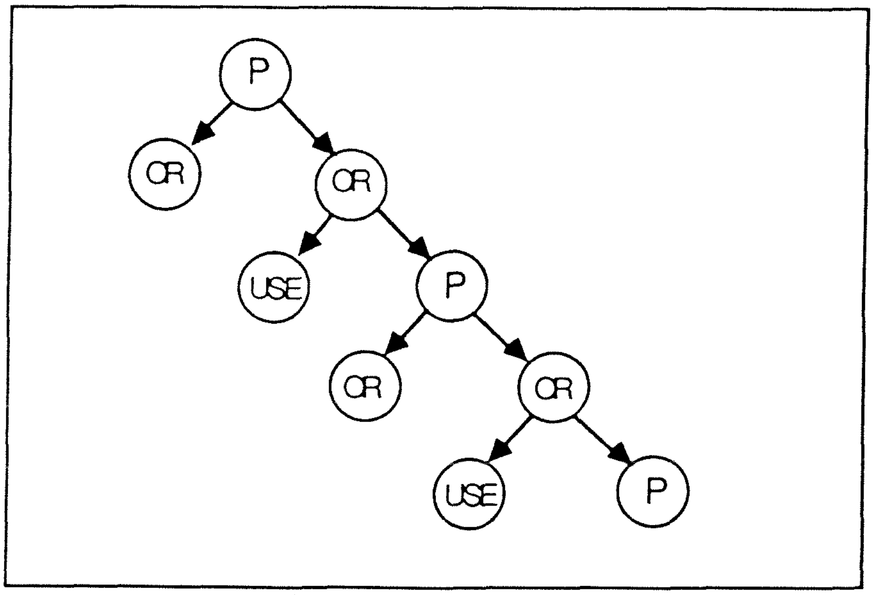

Figure 6.10. The Process Tree with Tail Recursion

The tail recursion optimization method proposed in the following aims to avoid the

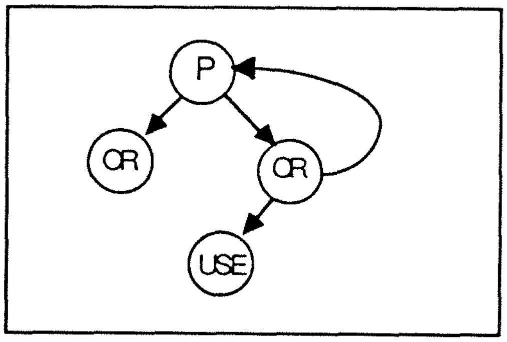

Figure 6.11. The Process Tree with Tail Recursion Optimization

long chain in the process tree as well as preserve the inherent parallelism in the clause. By observing the clauses with tail recursion, we can identify two different kinds of tail recursion, one with parallelism and one without.

6.3.1 Tail Recursion Optimization with Parallelism

Considering the case

$$p([X|L]) :- use(X), p(L). \tag{6.4}$$

The elements in the list are consumed one after another and the later iterations are independent of the previous iterations. Therefore, all the *use* literals can be invoked at once and the clause can be rewritten as

$$p([X1, X2, \ldots, Xn]) :- use(X1), use(X2), \ldots, use(Xn). \tag{6.5}$$

Notice that clause (6.5) is not a legal clause in CLP because n is unknown when p is defined. We now define an operator *map* which maps a predicate to all the elements of a list and concatenates the outputs into a list. Operator *map* has five arguments: the first argument is the predicate to be mapped, say p, the second and the third arguments are the input and the output lists, respectively, and the fourth and the fifth arguments are the positions of the input argument and the output argument of p. If the fifth argument of *map* is 0, then p generates no outputs. Operator *map* is similar to the MAPCAR function in Lisp.

Operator *map* succeeds when all the p's applied to the elements of the list succeed. The output of *map* is the concatenation of the output values of all the p's. Otherwise, *map* fails.

With *map*, (6.4) can be rewritten to

$$p(L) :- map(use(X), L, \_, 1, 0). \tag{6.6}$$

We propose to build an operator *map* in each OR process, which will expand (6.6) to (6.5), i.e., to create n AND processes for predicate *use* and assign one element of the list L to each process. The operator also concatenates the outputs from the AND processes in case the AND processes generate any result. Figure 6.12 shows the process tree of $p(L)$ with tail recursion optimization. One restriction for *map* is that it allows only one input argument and at most one output argument in the applied predicate. With this built-in operator *map*, we can get the most AND parallelism with the least overhead.

Another type of tail recursion which can be optimized by *map* operator is shown in (6.7).

$$\begin{aligned} &p([X|L]) :- test(X). \\ &p([X|L]) :- \neg test(X) \rightarrow p(L?). \end{aligned} \tag{6.7}$$

This program is equivalent to a *while* loop under the condition $test(X)$. The goal $p(L)$ succeeds if there exists at least one element in L that satisfies predicate *test*. In other words, the goal $p(L)$ fails if all the elements in L fail to satisfy *test*. Therefore, program (6.7) can be translated into a clause with *map* as follows:

$$p(L) :- \neg map(\neg test(X), L, \_, 1, 0). \tag{6.8}$$

The following *member* predicate is an example of this case:

$$\begin{aligned} &member(X, [X|L]). \\ &member(X, [Y|L]) :- X \neq Y \rightarrow member(X, L). \end{aligned}$$

Predicate *member* checks to see if X equals to any element in the list, which can be translated into:

$$member[X, L] :- \neg map(\neq (X, Y), L, \_, 2, 0).$$

Negation as Failure

In (6.8), a negation operator is applied to *map*. Negation is interpreted as "failure of proving a goal" in our Model. If an AND process holds a negated goal, say $\neg g$, it first resolves g by spawning OR processes for each definition of g. As soon as any one of its OR descendants returns a solution, the AND process fails; a *fail* message is sent to its father node and a *kill* message is sent to all its active descendants. The AND process succeeds if all its OR descendants fail. The process then sends a *binding*(*true*) message to its father node and terminates.

It is important to be aware that negation does not bind variables. Therefore, a negated literal in a clause body should not have any output variables. This implies that a negated literal can be fired only after all its variable arguments have been annotated as inputs during the execution of the ordering algorithm.

The negation operation before *map* in (6.8) is applied in an OR process. It is true when *map* is false; i.e., one of the AND descendants returns *fail*. It is false when *map* is true; i.e., all the AND descendants succeed. Like the negation in an AND process, it does not return any bindings. Therefore, the third argument of *map* should be null.

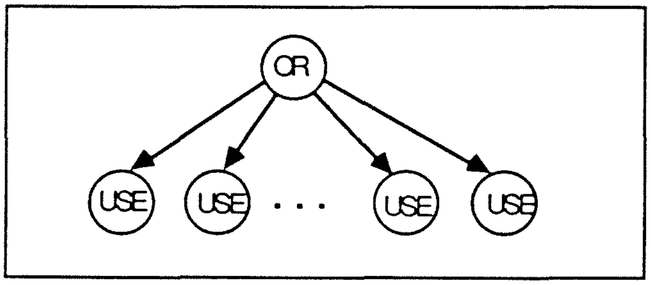

Figure 6.12. The Process Tree with Tail Recursion Optimization

Matrix multiplication is a good example of tail recursion with parallelism. With the *map* operation, the program is shown in Figure 6.13.

```
% To multiply two matrices, transpose the second, then form all inner products.
mm(A,B,C):-transpose(B,BT!),mmt(A,BT?,C).

% Multiply all rows of A with entire matrix B
mmt(A,B,C):-map(mmc(X,B,Y),A,C,1,3).

%Multiple all columns of B with row A.
mmc(A,B,C):-map(ip(A,X,0,Y),B,C,2,3).

%Form the inner product of two vectors.
ip([A1|A],[B1|B],S,C):-S1 is S+A1*B1,ip(A,B,S1?,C).
ip([],[],S,S).

% To transpose a matrix, call ''column'' to divide it into two parts:
% the first column and the rest of the columns; then transpose the
% rest.
transpose([[]|_],[]).
transpose(M,[C1|Cn]):-columns(M,C1,Rest!),transpose(Rest?,Cn).

columns([],[],[]).
columns([[C11|C1n]|C],[C11|X],[C1n|Y]):-columns(C,X,Y).
```

Figure 6.13. Matrix Multiplication Program

The tail recursion in predicate *ip* is not able to be translated into *map* because the second literal *ip* in the clause body needs the partial sum derived from the first literal so that the two literals are dependent of each other, neither does predicate *transpose* because of the shared variable *Rest* between the two literals in the body. Predicate *columns* cannot be translated into *map* either because *columns* produces two outputs while *map* allows only one output.

6.3.2 Tail Recursion Optimization without Parallelism

The predicates *ip* and *columns* are examples of tail recursion without parallelism. More specifically, the following two clauses can qualify in this category:

$$p([X|L], M) :- use(X, M, N), p(L, N?). \tag{6.9}$$

and

$$p([X|L], \ldots) :- p(L, \ldots). \tag{6.10}$$

The first case has no parallelism because the later iteration depends on the value N generated by the previous iteration; neither does the second case because there is no computation at all in this clause. (The output is generated in the unification.)

With this type of tail recursion, we can use the optimization technique shown in Figure 6.11, i.e., reuse the AND process at the top of the subtree and form a loop. Since the list argument in a tail recursive clause is usually a stream variable, this optimization scheme works very well together with the implementation of stream parallelism. The functions of the AND process and the OR process in the basic Sync Model have to be modified as follows:

1. An OR process that contains a tail recursive clause in the forms of (6.9) or (6.10) initiates the optimization. It performs the unification and the ordering algorithm as usual. Afterwards, it spawns one AND process for each of the literals in the clause body except the last one, i.e., $p(L, \ldots)$. It waits until all the descendants successfully terminate and receives all the variable bindings from the descendants. If there is any input variable in the arguments of p, the OR process binds the variables to the values received

from its descendants and sends the remaining variable bindings and $p(L, \ldots)$ to its father AND process and terminates.

2. When an AND process receives a goal $p(L, \ldots)$ from one of its OR descendants, it is notified that this AND process is to be used to solve the new goal. The AND process inspects all the variables in the new goal and adds the new variables to its channel table. The AND process then checks to see if all its OR descendants terminate. If so, the AND process discards the old goal and starts the new goal. The AND process terminates when all its OR descendants terminate and there is no new goal received from its descendant.

Take predicate ip as an example; suppose we want to compute the inner product of two vectors [1,2,3] and [3,2,1], $ip([1,2,3],[3,2,1],0,C)$ is called and an AND process is created for this goal. The process tree is constructed as shown in Figure 6.14. Two OR processes are spawned by the AND process and the first OR process succeeds in unification. The OR process detects that the clause it contains is tail recursive without parallelism. The two literals in the clause body contain a shared variable $S1$. The OR process spawns an AND process to calculate $S1$ and keeps the second AND literal. When the binding $\langle S1, 3\rangle$ is received by the OR process, $S1$ in the second AND literal is replaced by 3 and $ip([2,3],[2,1],3,C)$ is sent back to the top AND process; then the OR process terminates. The top AND process receives the new goal and detects that both its descendants have terminated, then, it starts to resolve the new goal $ip([2,3],[2,1],3,C)$. Notice that variable C is already in its channel table. The execution proceeds until the top AND process receives a new goal $ip([],[],10,C)$ from its first descendant. In execution of $ip([],[],10,C)$, the second OR descendant succeeds and returns a binding $\langle C, 10\rangle$. The top AND process receives and sends out $\langle C, 10\rangle$ finally and terminates itself.

The above example has no stream variables. The list arguments of ip have to be transmitted from the top AND process to the OR descendants repeatedly. With stream parallelism, the list elements are sent down one by one and the communication overhead is greatly reduced.

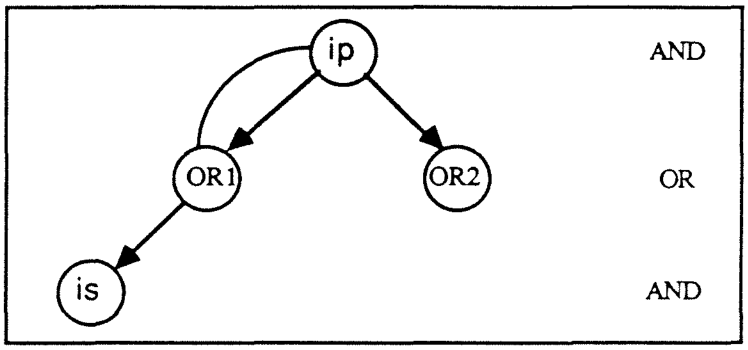

Figure 6.14. An Example of Tail Recursion Optimization without Parallelism

6.4 Nondeterministic and Deterministic Modes

Our Sync Model is adequate for executing deterministic programs as well as nondeterministic programs. When a nondeterministic program is executed, all the solutions will be returned. Our model is designed to handle a multiple solution stream along each channel. Therefore, it is less efficient for a deterministic program because there is at most one solution along each channel in the process tree. If we restrict ourselves to a deterministic program, our model can be greatly simplified. Since there is a big group of applications falling into the category of deterministic programs, it is beneficial to have two different programming modes in our model: the Nondeterministic mode and the Deterministic mode.

In the Deterministic mode, there is at most one solution along each channel of the process tree. As a consequence, we don't need to append an "END" signal to a solution at a leaf process and detect multiple paths in the data flow graph in the ordering algorithm. Incidentally, the merge algorithm can be eliminated in both the AND processes and the OR processes since there are no multiple solutions to be merged. In the Deterministic mode, the communication overhead is largely reduced because all the synchronization signals such as Sync signals and END signals are eliminated. The computation overhead is also reduced because the merge operation and the most time-consuming part of the ordering algorithm, the detection of the multiple paths, are no longer necessary.

There are many situations in which we only need one solution for a nondeterministic program. For instance, a map coloring problem is a nondeterministic problem. If there exists one solution for that problem, there are many symmetric solutions that can be derived by changing the color assignment of the first solution to another color permutation. For this kind of problem, we are interested in the existence of the solutions; thus, it is not necessary to compute all the solutions. Since our model always returns all the solutions, we need some mechanism to stop the computation when the first solution is derived. The solution is simple. When the root process returns the first solution to the outside interface, a *kill* signal is issued to the root and propagated throughout the process tree to interrupt all the ongoing computations. There is no extra cost because the interrupt mechanism is already implemented in our Model.

6.5 Summary

In this chapter, we first modify the Sync Model so that it can handle partially instantiated terms in variable bindings. Then, we extend the Sync Model to implement stream parallelism and tail recursion optimization.

In our model, the data flow graph of AND literals is changed dynamically by adding dynamic links when a variable's producer binds the variable to a partially instantiated term. With dynamic links, the "one producer per variable" rule is preserved without reconstruction of the data flow graph with the ordering algorithm. A new type of message, the *channel* message, is used to transmit the channel information for dynamic links. The *channel* message is handled by the MERGE_AND routine.

Potential stream parallelism can be detected by our Model through the identification of the stream variables in the program. Special communication channels, i.e., the stream channels, directed both horizontally and vertically are constructed to transmit the partial values of the stream variables. With stream parallelism, our model is able to handle unbounded lists and to achieve a more efficient computation.

Tail recursion is categorized into two different forms: one with parallelism and one without. An *map* operator is proposed to handle tail recursion optimization with parallelism. This operator residing in an OR process creates a set of parallel AND processes and each process computes one element of the list. This approach retains the parallelism of the clause and avoids a long chain of processes as well. To implement tail recursion optimization without parallelism, the AND process is repeatedly used to resolve the goals which are recursively called. Combined with stream parallelism, this implementation is clean and simple, while no stack or state variables are required.

Lastly, we propose to separate the operation modes into Nondeterministic and Deterministic modes to further increase the efficiency of our Model. In the Deterministic mode, the merge algorithm, the synchronization signals and the detection of multiple paths are no longer necessary and the model is extensively simplified.

Chapter 7

The Sneptree

7.1 Introduction

The choice of interconnection networks is one of the main concerns in designing an ensemble machine. The mapping of a distributed computation onto a fixed processor network is an important issue in designing a parallel software system. To partition a problem into smaller subproblems with minimal interfaces is crucial in designing a parallel algorithm. In the previous chapters, we have seen that logic programming is a natural way to express the parallelism within a problem by distributing the computations into a dynamic tree of processes. In this chapter, we present a new interconnection network, the *Sneptree* [49], and we show that the Sneptree is an ideal network for computations with a dynamic tree as computation graph.

The Sneptree is a class of augmented binary trees with identical nodes. Each node, including the root node and the leaf node, has four links. The links are connected such that a complete binary tree of arbitrary size can be mapped onto the Sneptree optimally. Besides, some connection patterns of the Sneptree are regular and symmetric and hence well suited for VLSI implementation.

In Section 7.2, the definitions of the Sneptree and the Cyclic Sneptree are given and different connection patterns are presented. The mapping of a complete binary tree onto the Sneptree is proven to be optimal in Section 7.3. Like a binary tree, the Sneptree can be laid out into an H-structure plane nicely. Section 7.4 presents a recursive method to construct the H-structure Sneptree. The comparison of the Sneptree and other similar networks is discussed in the conclusion. In Chapter 8, we present a leaf node routing algorithm for the Cyclic Sneptree, which takes advantage of the extra links in the Sneptree.

7.2 Definition of the Sneptree

Definition: An *n-level Sneptree* is a complete binary tree of $2^n - 1$ nodes, links directed from root to leaves, augmented with 2^n additional *Snep* links directed out of the leaves, such that each node has 4 incident links: 2 directed in and 2 directed out. Each node in the tree has an incoming Sneplink, except for the root, which has 2 incoming Sneplinks.

Notice that the Sneptree is defined to be a directed graph here for easier understanding. In the real implementation, the links should be bidirectional. Furthermore, we call the outgoing link which points to the left descendant of a node the *left link* and the one pointing to the right descendant the *right link.*

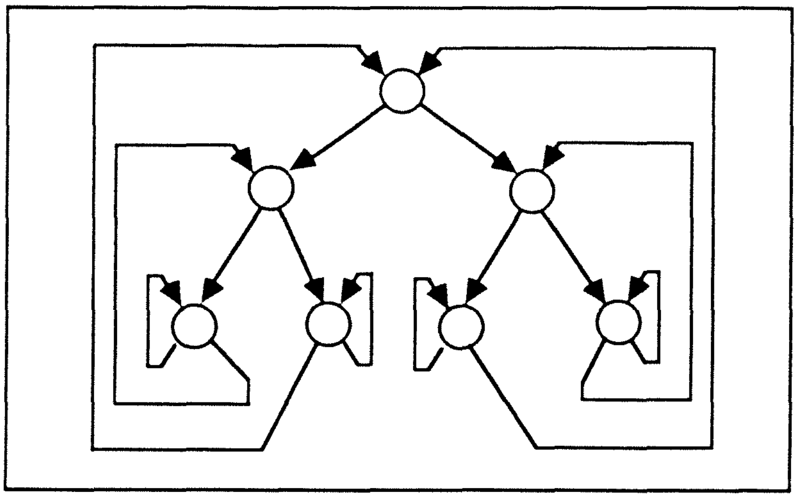

Figure 7.1. A Three–level Sneptree

There are many possible ways to connect the 2^n Sneplinks. One example of a three-level Sneptree is shown in Figure 7.1. This connection is planar, symmetric and extensible. Nevertheless, it is not of particular interest because it renders a very unbalanced mapping for a highly unbalanced binary tree, such as a left-skewed tree. Another type of Sneptree whose Sneplinks are connected to form two spanning cycles (i.e., Hamiltonian Cycles) [20] renders an optimal mapping for a left(right)- skewed tree of any size (i.e., a linear array).

This special type of Sneptree is called a *Cyclic Sneptree.*

Definition: A *Cyclic Sneptree* is a Sneptree containing two link-disjoint spanning cycles. The *left cycle* contains only left links and the *right cycle* contains only right links.

Theorem 7.1. *There are* $[(2^{n-1}-1)!]^2$ *connection patterns for the n–level Cyclic Sneptree.*

Proof: In an n–level binary tree, there are 2^{n-1} leaf nodes as well as 2^{n-1} extra left outgoing links. Each leaf node is the ending node of a path containing left outgoing links only. The starting node of such a path is a node that is not a left descendant of any other node. There exist 2^{n-1} such paths. All the paths are distinct and they don't share any nodes or links. Moreover, the union of the paths covers all the nodes in the binary tree. A cycle is constructed by combining all the paths in the way of connecting the left outgoing link of the ending node in one path to the incoming link of the starting node of another path. This cycle now covers all the nodes. There are $(2^{n-1}-1)!$ such cycles (i.e., left spanning cycles). Similarly, there are the same number of right spanning cycles. Since the connections of the left spanning cycles and the right spanning cycles are independent, there are $[(2^{n-1}-1)!]^2$ connections which result in a Cyclic Sneptree. ∎

In these $[(2^{n-1}-1)!]^2$ connection patterns, many of them are isomorphic because the left and the right links are indistinguishable in practice.

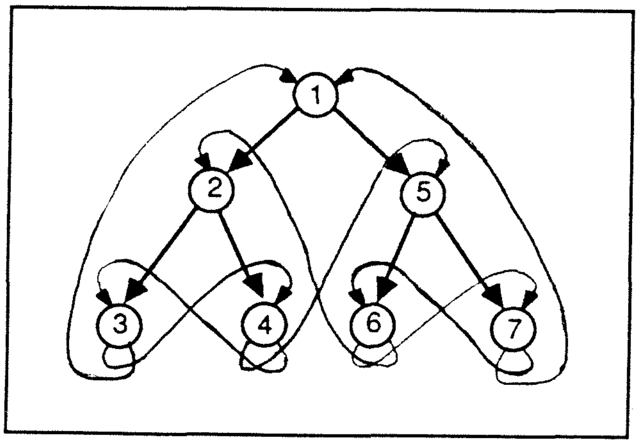

Figure 7.2. A Three–level Cyclic Sneptree

Figure 7.2 shows one connection pattern of Cyclic Sneptree. The numbers attached to the nodes show the node ordering in the left spanning cycle. Symmetrically, the right spanning cycles of Figure 7.2 can be represented by node sequences (1,5,7,6,2,4,3,1). Such a connection pattern is regular and symmetric and it can be generated recursively from the smaller structures. Although this connection pattern is not planar, the two crossing Sneplinks between two adjacent subtrees make it possible to extend one subtree to the other subtree. Therefore, the result of mapping an unbalanced binary tree onto Figure 7.2 is better than the result of mapping the same tree onto the Sneptree in Figure 7.1. This particular Cyclic Sneptree is chosen due to its regularity and extensibility, which are crucial properties for VLSI implementation. Another connection pattern with the same properties is compared in the conclusion. We show later that the connection we choose here is better.

7.3 Mapping of a Binary Tree onto a Sneptree

It is important to mention that the mapping from a complete binary tree onto a Sneptree is independent of the connection patterns of the Sneptree. In other words, no matter how the Sneplinks are connected in a Sneptree, the mapping of a complete binary tree is always optimal. This is not true for an incomplete binary tree. The performance of the mapping of an unbalanced tree is affected by the connection pattern of the Sneptree.

Before describing the mapping performance, we shall first define the *computation graph* and the *implementation graph.* According to [31], the computation graph represents the structure of the distributed computation, and the implementation graph represents the network topology of the parallel machine. *Cell* and *node* are the names for a vertex of the computation graph and the implementation graph, respectively. A *legal mapping* from a computation graph to an implementation graph maps adjacent cells onto adjacent nodes. Moreover, an *optimal mapping* is defined as a legal mapping such that the number of cells contained in a single node differs by at most one from the average number of cells in each node of the Sneptree. The legal mapping defined here is rather restricted. Such a mapping may or may not exist for arbitrary computation graphs and implementation graphs. If the neighboring cells are allowed to be mapped onto non-adjacent nodes, a cost function, which is the sum of the distance of the neighboring cells in the mapping, is usually defined to measure the performance of the mapping. The legal mapping we defined here is actually a mapping with cost function equal to one.

From now on, we assume the computation graph is an m–level complete binary tree, the implementation graph is an n–level Sneptree and m $\geq$ n. Again, a cell denotes a node in the binary tree and a node denotes a node in the Sneptree. There are two important measures for the performance of this particular mapping. The first one is the total number of cells mapped onto one single node, which indicates the total work load of each node. The second one is the number of cells of the same height in the binary tree mapped onto one single node, which is important when the computation wavefront goes downward and upward in the tree so that only the nodes at one particular level are active at a time. In the following, we are going to show that both measures are minimal in mapping a complete binary tree onto a Sneptree. Therefore, the mapping is optimal.

In an n–level complete binary tree, the root is at level 0 and the leaves at level (n−1). The height of a node is defined to be the minimal distance between this node and the leaves. The height of a tree is the distance between the root and the leaves, i.e., (n−1). The above definitions also apply to an n–level Sneptree.

It is obvious that there exists a legal mapping from a complete binary tree onto a Sneptree. One such mapping maps the root of the binary tree onto the root of the Sneptree and the two children of a cell onto the two direct descendants of a node. With such a mapping scheme, we find that a complete binary tree can be mapped onto a Sneptree optimally from the following theorems.

Theorem 7.2. *All the nodes in the Sneptree contain the same number of cells of one particular level, say k, except that each node at the* $(k \bmod n)$*-th level contains one more cell, when mapping an m-level complete binary tree onto a n-level Sneptree,* $m \geq n$ *and* $0 \leq k < m$.

Proof: If $k < n$, one cell will be mapped onto one node at the k-th level in the Sneptree. In other words, all the nodes in the Sneptree contain no cells at the k-th level of the binary tree except the nodes at the k-th level, which contain one cell in each node. For $k \geq n$, the theorem can be proven by observing the construction of the Sneptree. A node at the j-th level of the Sneptree has two direct ancestors; one is its father at $(j-1)$-th level, and the other is a leaf, i.e., a node at the $(n-1)$-th level. Moreover, the number of cells of level k that are mapped onto this node is the sum of the cells of level $(k-1)$ that are located in its direct ancestors. In other words,

$$T_k(j) = T_{k-1}(j-1) + T_{k-1}(n-1), \qquad j > 0 \tag{7.1}$$

where $T_k(j)$ is the total number of cells at the k-th level of the binary tree, which are mapped onto one node located at the j-th level of the Sneptree for $0 \leq k < m$ and $0 \leq j < n$.

The root node, i.e., the node at the 0-th level, has no upper level and its two direct ancestors are both from the bottom level, i.e., the (n−1)-th level. Combined with Eq.(7.1), $T_k(j)$ can be recursively defined by

$$T_k(j) = \begin{cases} T_{k-1}((j-1) \bmod n) + T_{k-1}(n-1), & \text{if } n \leq k < m, 0 \leq j < n, \\ 0, & \text{if } 0 \leq k < n, 0 \leq j < n \wedge j \neq k; \\ 1, & \text{if } 0 \leq k < n, 0 \leq j < n \wedge j = k. \end{cases} \tag{7.2}$$

By induction, assume that the theorem holds for any $k \geq n$; i.e., all the nodes in the Sneptree contain the same number of cells of level k, except that each node at ($k \bmod n$)-level contains one more cell. Let $k = q \times n + r$, then $T_k(r) = T_k(j) + 1$, for all j and $j \neq r$. We now prove that the theorem holds for $(k+1)$. From Eq.(7.2), when $r \neq n-1$,

$$T_{k+1}((k+1) \bmod n) = T_{k+1}(r+1) = T_k(r) + T_k(n-1) = 2 \times T_k(n-1) + 1, \quad \text{and}$$

$$T_{k+1}(j) = T_k((j-1) \bmod n) + T_k(n-1) = 2 \times T_k(n-1), \quad j \neq (k+1) \bmod n.$$

If $r = n-1$,

$$T_{k+1}((k+1) \bmod n) = T_{k+1}(0) = T_k(n-1) + T_k(n-1) = 2 \times T_k(j) + 2, \quad \text{and}$$

$$T_{k+1}(j) = T_k((j-1) \bmod n) + T_k(n-1) = 2 \times T_k(j) + 1, \quad j \neq (k+1) \bmod n.$$

Therefore, $T_{k+1}((k+1) \bmod n) = T_{k+1}(j) + 1, j \neq (k+1) \bmod n$, holds for any k.

By induction, the theorem holds. ∎

Theorem 7.3. *All the nodes at the top ($m \bmod n$) levels of the Sneptree contain the same number of cells. Similarly, the rest of the nodes also contain the same number of cells, and the number is one fewer than that in the top level nodes, when mapping an m-level complete binary tree onto an n-level Sneptree, $m \geq n$.*

Proof: Let $T(j)$ be the total number of cells mapped onto one node at the j-th level of the Sneptree; i.e., $T(j) = \sum_{k=0}^{m-1} T_k(j)$. Let $m = q \times n + r$ and consider a node at one particular level j. Such a node contains one more cell at $(n+j)$-th, $(2n+j)$-th, ..., and $(q \times n + j)$-th levels, respectively, than the nodes not in level j, when $j \le r$. In other words, there are q such levels in the binary tree, in which one extra cell is assigned to the nodes at level j, for $j \le r$. For $j > r$, there exist only $q-1$ such levels. Therefore, we can conclude that all the nodes at the top $r = (m \bmod n)$ levels of the Sneptree contain the same number of cells. Likewise, the rest of the nodes also contain the same number of cells, and the number of cells is one fewer than that of the top level nodes. ∎

Corollary 7.4. *For $k = n, n+1, \ldots, m-1$*

$$T_k(j) = \begin{cases} \left\lfloor \dfrac{2^k}{2^n - 1} \right\rfloor, & \text{if } 0 \le j \le n-1 \text{ and } j \ne k \bmod n \\[2ex] \left\lfloor \dfrac{2^k}{2^n - 1} \right\rfloor + 1, & j = k \bmod n. \end{cases}$$

and

$$T(j) = \begin{cases} \left\lfloor \dfrac{2^m - 1}{2^n - 1} \right\rfloor + 1, & \text{for } 0 \le j < (m \bmod n) \\[2ex] \left\lfloor \dfrac{2^m - 1}{2^n - 1} \right\rfloor, & \text{for } (m \bmod n) \le j < n. \end{cases}$$

The above corollary can be derived immediately from Theorem 7.2 and Theorem 7.3. We now get to the conclusion which has been addressed at the beginning of this section.

Theorem 7.5. *The mapping of a complete binary tree onto an arbitrary Sneptree is optimal.*

From the above discussion, it is clear that an arbitrary size complete binary tree can be mapped onto a Sneptree optimally no matter how it is connected. However, the performance of mapping an unbalanced binary tree is dependent on the connection pattern of the Sneptree. The Cyclic Sneptree has the best performance on mapping a left-skewed or right-skewed tree. Another type of the Sneptree, called the "Exchange Sneptree," in which the Sneplinks are connected to provide full connection between the two halves of the Sneptree, performs better on mapping an arbitrary unbalanced binary tree. More discussions about the Exchange Sneptree can be found in Section 7.5.2.

Theorem 7.6. *A left or right skewed tree of any size can be mapped onto a Cyclic Sneptree optimally.*

Proof: The theorem is true from the definition of the cyclic Sneptree. ∎

Remark A linear array of any length can be mapped onto the cyclic Sneptree optimally if we map the linear array onto the left cycle or the right cycle of the Cyclic Sneptree.

7.4 Layout of a Cyclic Sneptree onto an H-structure Plane

In this section, we discuss how to lay out a Cyclic Sneptree (shown in Figure 7.2) on a plane. From now on, we call the Cyclic Sneptree (such as in Figure 7.2) "Sneptree," since all the discussions in this section are based on this particular connection pattern. The H-structure layout for a binary tree [34] is modified and adapted to layout a Sneptree. The recursive rule to generate the layout is described. Also, the number of crossings, the area, and the length of the longest wires are analyzed. Finally, we will present a way to extend the size of the Sneptree by connecting two identical smaller Sneptrees.

7.4.1 Recursive Generation of H-structure Layout

Like the binary tree, the Sneptree can be laid out into an H-structure plane. Because of the Sneplinks in the Sneptree, it is not that straightforward to build the H-structure Sneptree. The major concern is to minimize the number of crossings in the layout and keep the length of Sneplinks as short as possible. With these two criteria in mind, a recursive construction algorithm is designed.

The n–level Cyclic Sneptree can be constructed recursively into an H–structure layout with two given basic three–level H–Sneptrees, A_3 and B_3 (Figure 7.3). In Figure 7.3, the node numbering is compatible with that of Figure 7.2. The dangling arrows out of node 3 and node 7 are the two links incident to the root in a regular 3-level Cyclic Sneptree. These two links are dangling in order to extend to bigger structures.

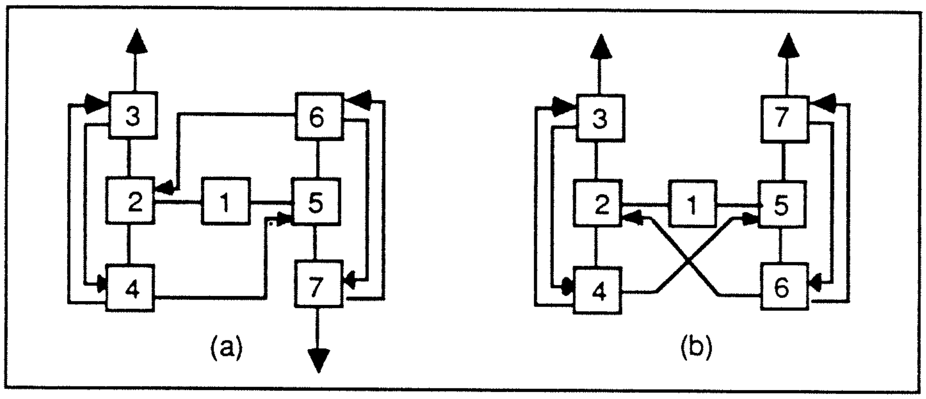

Figure 7.3. Two Basic Three–level H–Sneptrees

We define two basic operations on a layout G:

(a) mirror along x axis : G^x

(b) mirror along y axis : G^y.

The recursive rules are as follows:

1. Construct two 3-level H-structure Sneptrees A_3 and B_3 as shown in Figure 7.3. A_3 is the one we intend to construct, and B_3 is an auxiliary graph to be used in constructing bigger Sneptrees. Now we would like to construct A_n for all $n \geq 3$ and B_n for $n \geq 3$ and n odd.

2. Given two k–level H-structure Sneptrees, A_k and B_k, for $k \geq 3$ and k is odd, the (k+1)–level and (k+2)–level H-structure Sneptrees can be constructed as shown in Figure 7.4. A (k+1)–level Sneptree, A_{k+1}, can be constructed by two k–level subtrees, namely, A_k and B_k. And a (k+2)–level Sneptree, A_{k+2}, can be constructed by four k–level Sneptrees, A_k, B_k, B_k^y and A_k^x. The auxiliary (k+2)–level Sneptree, B_{k+2}, can be constructed by A_k^x, B_k, A_k^y and B_k.

For example, a 4–level Sneptree is constructed by connecting A_3 and B_3 to an extra node and by directing the Sneplinks as shown in Figure 7.5.a. Notice that A_3 is planar and B_3 has one crossing. A_4 has five crossings due to the introduction of new links, including the two links incident to the root, shown as dotted lines in Figure 7.5.a. The dotted arrows into the root node should be connected to the two dangling links coming out of the leaf nodes to make it a complete 4-level Sneptree. A 5–level Sneptree can be constructed as in

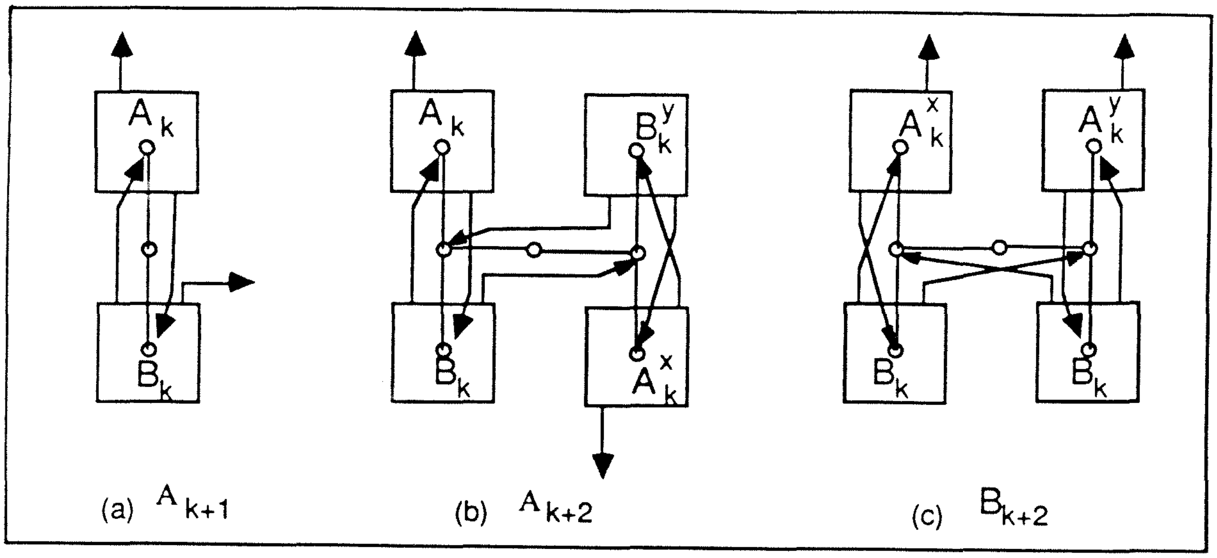

Figure 7.4. Construction of A_{k+1}, A_{k+2} and B_{k+2} Using A_k and B_k

Figure 7.5.b. There are in total eleven crossings in the layout: five in the left half of the graph, which is exactly A_4 except for the two links of the root coming out to the right; four in the right half as shown in the figure, and two from the incoming links (dotted lines) of the root in the middle of Figure 7.5.b.

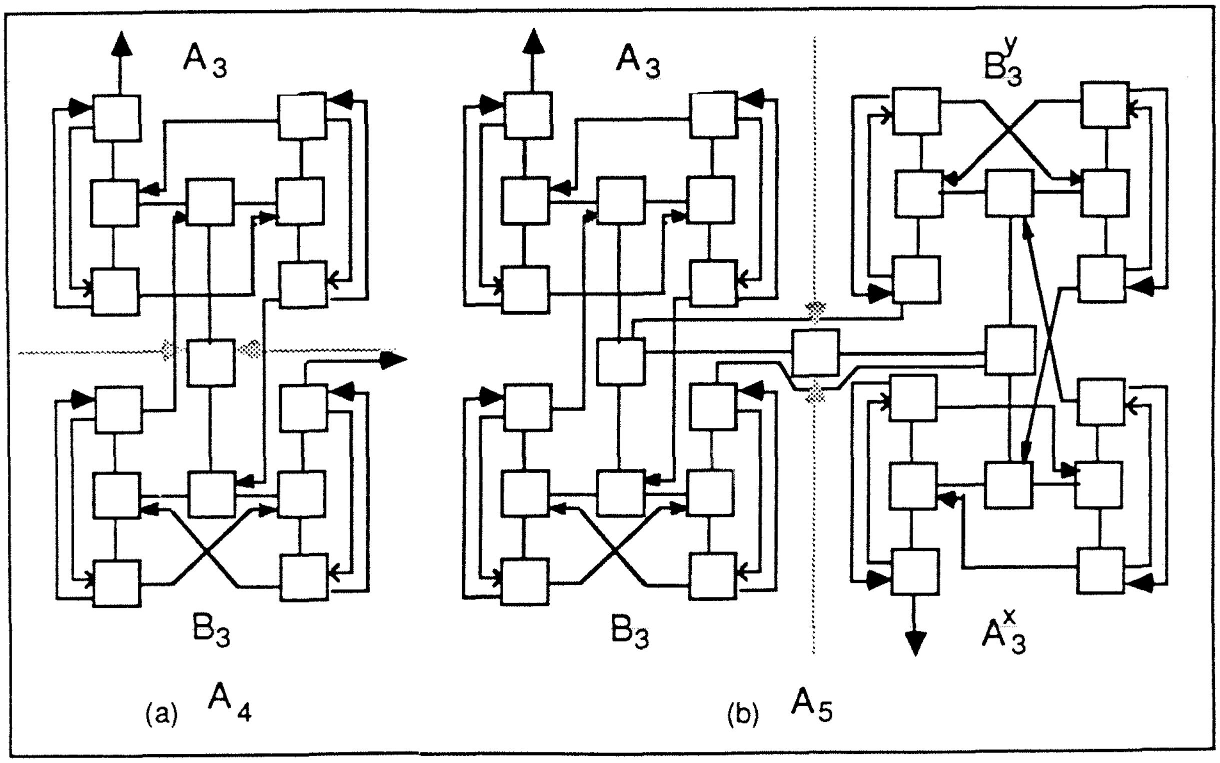

Figure 7.5. Construction of A_4, A_5 Using A_3 and B_3

Since the Cyclic Sneptree is not planar and the Sneplinks are not of constant length, we would like to know the number of crossings and the maximum length of the Sneplinks.

Theorem 7.7 gives the number of crossings in A_n and B_n. B_n has one more crossing than A_n and both figures are approximately 3/8 of the total number of nodes in the Sneptree. The two crossings introduced by the incoming links of the root node in A_n are not counted.

Theorem 7.7. *The number of crossings in A_n is $3 \times (2^{n-3} - 1)$ and the number of crossings in graph B_n is $3 \times (2^{n-3} - 1) + 1$.*

Proof: By induction. Let $C(A_n)$ and $C(B_n)$ be the number of crossings in graph A_n and B_n, respectively.

For n=3, $C(A_3) = 0$ and $C(B_3) = 1$, which is true from Figure 7.3.

Assume k to be odd, $C(A_k) = 3 \times (2^{k-3} - 1)$ and $C(B_k) = 3 \times (2^{k-3} - 1) + 1$; then we want to prove that the equations hold for A_{k+1}, A_{k+2}, and B_{k+2}.

Observing the construction in Figure 7.4, two extra links incident to the root node of A_k will cross over the internal link of A_k from the left bottom to the right middle. No extra crossings are introduced in the lower half, i.e., B_k, since two horizontal links (the links go from the left half to the right half) are both beneath the root of B_k (see Figure 7.3.b and Figure 7.4.c). Hence, the number of crossings in A_{k+1} is

$$\begin{aligned} C(A_{k+1}) &= C(A_k) + C(B_k) + 2 \\ &= 3 \times (2^{k-3} - 1) \times 2 + 3 \\ &= 3 \times (2^{k-2} - 2 + 1) \\ &= 3 \times (2^{(k+1)-3} - 1). \end{aligned}$$

To construct A_{k+2}, the left half of A_{k+2} is exactly A_{k+1} (see Figure 7.4.b). The right half contains graph B_k mirroring along y direction and graph A_k mirroring along x direction, which have the same number of crossings as B_k and A_k, respectively. Similarly, two extra crossings appear where the two extra links are incident to the root of A_k^x. Another vertical crossing is shown in Figure 7.4.b, when two links go from the upper half to the lower half and vice versa. Finally, the extra horizontal links add two more crossings to the left part of this picture. The equation turns out to be:

$$\begin{aligned} C(A_{k+2}) &= C(A_{k+1}) + C(A_k) + C(B_k) + 2 + 1 + 2 \\ &= 3 \times (2^{k-2} - 1) + 3 \times (2^{k-2} - 2) + 6 \\ &= 3 \times (2^{(k+2)-3} - 1). \end{aligned}$$

Comparing B_{k+2} with A_{k+2} in Figure 7.4.c and 7.5.b, B_{k+2} has one more crossing than A_{k+2} due to the crossover of the two horizontal links in the middle of the graph. Therefore,

$$\begin{aligned} B_{k+2} &= A_{k+2} + 1 \\ &= 3 \times (2^{(k+2)-1} - 1) + 1. \end{aligned}$$

By induction, for all $n \geq 3$, the equations are true. ∎

Let us assume that any single node in the layout is a square with area a; i.e., each side of the node is $\sqrt{a}$ in length and the wire width is negligible compared to $\sqrt{a}$. The area of the H-structure Sneptree is a function of a and the size of the Sneptree, n. The zero wire width assumption is reasonable because the number of wires passing through any two adjacent nodes in the layout is bounded by a fixed number.

Theorem 7.8 gives the area of an n-level H-structure Sneptree. Since there are $2^n - 1$ processors in a n-level Sneptree, the total processor occupation ratio $(a \times (2^n - 1))/A_n$ has the lower bound 2/3 when n grows to infinity.

Theorem 7.8. *The layout area of an n-level H-structure Sneptree is*

$$A_n = \begin{cases} 3a \times (2^{n-1} - 2^{(n-2)/2}), & n \textit{ is even} \\ 3a \times (2^{n-1} - 2^{(n-3)/2}), & n \textit{ is odd} \end{cases}$$

Proof: By induction. From Figure 7.6, it is easy to see that the layout areas for 2-level and 3-level Sneptrees are $3 \times a$ and $9 \times a$, respectively, satisfying the equation above.

For n>3 and even, the layout is constructed by two (n-1) layouts vertically (see Figure 7.7.a). The root node is positioned in the middle of the two (n-1) layouts. It doesn't need extra space because there is enough room to put an extra node in the center column. Let the width and the length of the n-level H-structure Sneptree be W_n and L_n. For even n, L_n is the double of L_{n-1} and W_n is the same as W_{n-1} from the construction rule. For n is odd, the layout is constructed by two (n-1) layouts horizontally with the root in between (see

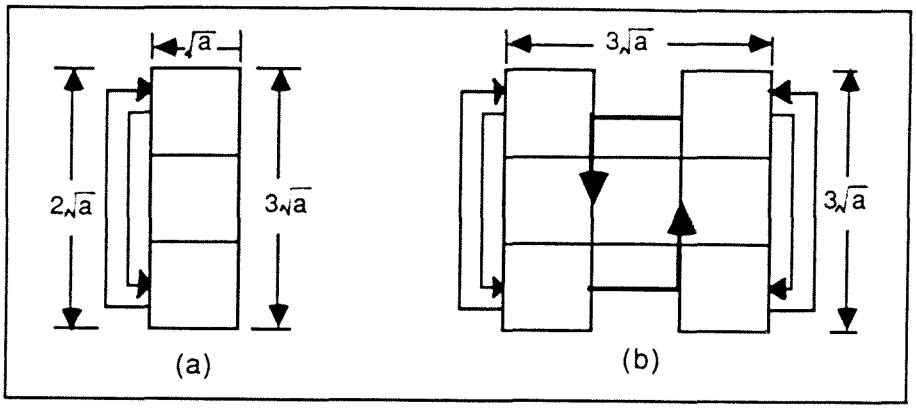

Figure 7.6. The Areas and the Longest Wires of the 2-level and 3-level H-structure Sneptrees

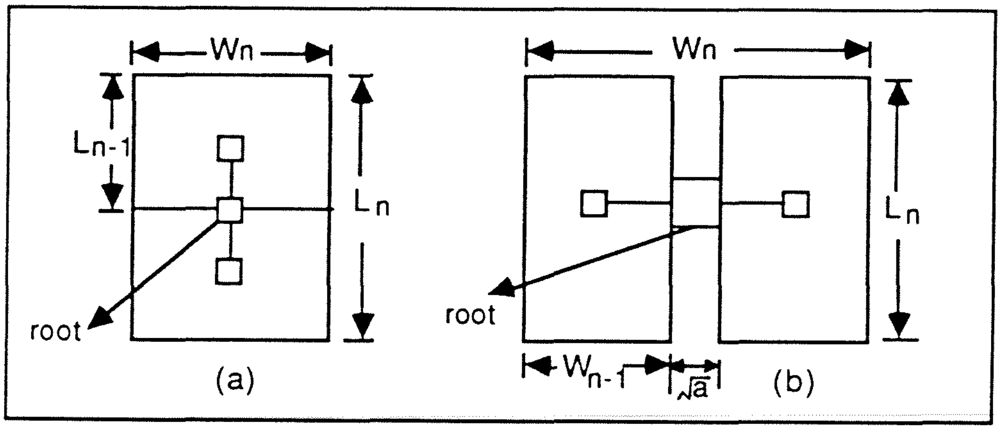

Figure 7.7. The Area of an n-level H-structure Sneptree

Figure 7.7.b). The root node is placed in an extra column because the rightmost column of the left half and the leftmost column of the right half are fully occupied and no room is left for this extra node. Hence, the width of the layout is $W_n = 2W_{n-1} + \sqrt{a}$ and L_n is the same as L_{n-1} for odd n. We now have the following two sets of recurrence functions. For even n:

$$L_n = 2 \times L_{n-1}$$

$$W_n = W_{n-1}$$

and for odd n:

$$L_n = L_{n-1}$$

$$W_n = 2W_{n-1} + \sqrt{a}$$

with initial condition

$$L_2 = 3\sqrt{a}$$

$$W_2 = \sqrt{a}.$$

Solving above recurrence for even n, we get

$$\begin{aligned} L_n &= 2 \times L_{n-2} \\ &= 2^2 \times L_{n-4} \\ &= \dots \\ &= 2^{n/2-1} \times L_2 \\ &= 3\sqrt{a} \times 2^{n/2-1} \end{aligned} \tag{7.3}$$

and

$$\begin{aligned} W_n = W_{n-1} &= 2 \times W_{n-2} + \sqrt{a} \\ &= 2^2 \times W_{n-4} + \sum_{i=0}^{1} 2^i \times \sqrt{a} \\ &= 2^{n/2-1} \times W_2 + \sum_{i=0}^{n/2-2} 2^i\sqrt{a} \\ &= \sqrt{a}(2^{n/2-1} + 2^{n/2-1} - 1) \\ &= \sqrt{a}(2^{n/2} - 1) \end{aligned} \tag{7.4}$$

so that the layout area for even n is

$$\begin{aligned} A_n &= L_n \times W_n \\ &= 3a \times (2^{n-1} - 2^{(n-2)/2}) \end{aligned}$$

and the layout area for odd n is

$$\begin{aligned} A_n &= L_n \times W_n \\ &= L_{n-1} \times (2W_{n-1} + \sqrt{a}) \\ &= 3\sqrt{a} \times 2^{n-1/2-1} \times (2\sqrt{a}(2^{n-1/2} - 1) + \sqrt{a}) \\ &= 3a \times (2^{n-1} - 2^{(n-3)/2}) \end{aligned}$$

∎

Furthermore, assume that the four links of each node may be pulled out of any side of the node. Two or more links may come out of the same side. The length of the wire connecting two nodes is the shortest distance from the center of one side of the source node to the center of the nearest side of the other node. The wire has to route around all the nodes in the way.

Theorem 7.9 shows that the length of the longest internal wire in an H-layout Sneptree is about 1/4 of the width of the layout. It is only $(3/2)\sqrt{a}$ longer than the longest wire of the H-layout binary tree of the same size.

Theorem 7.9. *The longest internal wires of an n-level H-structure Sneptree are the two wires connecting the root and two leaf nodes at the left and the right corners. The length is*

$$l_n = \begin{cases} \sqrt{a}(3 \times 2^{n/2-3} + 1/2), & n > 3 \text{ and even} \\ \sqrt{a}(2^{(n-3)/2} + 1/2), & n > 3 \text{ and odd} \\ 2\sqrt{a}, & n \leq 3 \end{cases}$$

Proof: By induction. For n=2, the longest wire is the one from one leaf node to the other leaf node, which is $2\sqrt{a}$ (Figure 7.6.a). For n=3, all the extra links are of equal length, i.e., $2\sqrt{a}$ (see Figure 7.6.b).

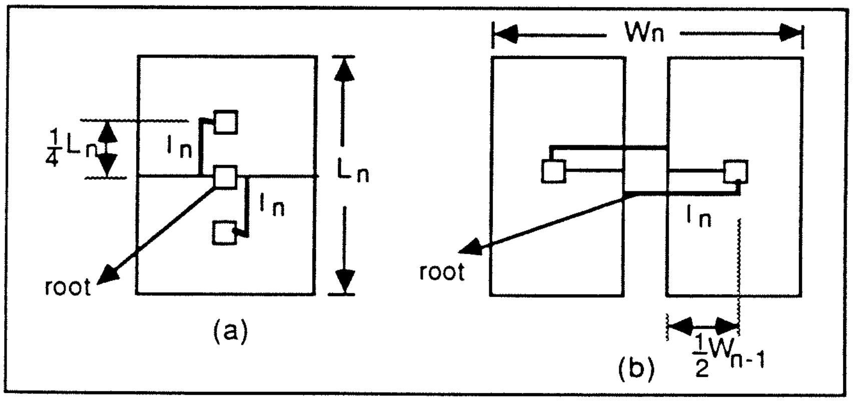

Figure 7.8. To Find the Longest Wires in Terms of L_n or W_n

For n>3 and even, the layout is constructed by two (n-1) layouts vertically (see Figure 7.8.a). The root node is positioned in the middle of the two (n-1) layouts. It doesn't need extra space because there is enough room to put an extra node in the center column. The two longest wires are shown as Figure 7.8.a. Let the width and the length of the n-level H-structure Sneptree be W_n and L_n; then the length of the longest wire is one quarter of L_n plus half of one side of the node. In other words,

$$l_n = \frac{1}{4}L_n + \frac{1}{2}\sqrt{a}.$$

Substituting (2.3) for L_n, we get

$$l_n = \sqrt{a}(3 \times 2^{n/2-3} + 1/2).$$

For n>3 and odd, the layout is constructed by two (n-1) layouts horizontally with the root in between (see Figure 7.8.b). The two longest wires are as shown in the figure.

Apparently, the length of the longest wire is

$$l_n = \frac{1}{2}W_{n-1} + \sqrt{a}.$$

Substituting (2.4) for W_{n-1}, we get

$$l_n = \sqrt{a}(2^{(n-3)/2} + 1/2).$$

This completes the proof. ∎

7.4.2 Extension of the Sneptree

From the above discussion, it is clear that we can lay out a Sneptree of any size onto a single chip as long as the chip capacity is not exceeded. Here we present a method to extend the Sneptree by connecting two identical H-structure Sneptrees. This method is extended from the recursive construction technique of binary tree from a single chip proposed in [26].

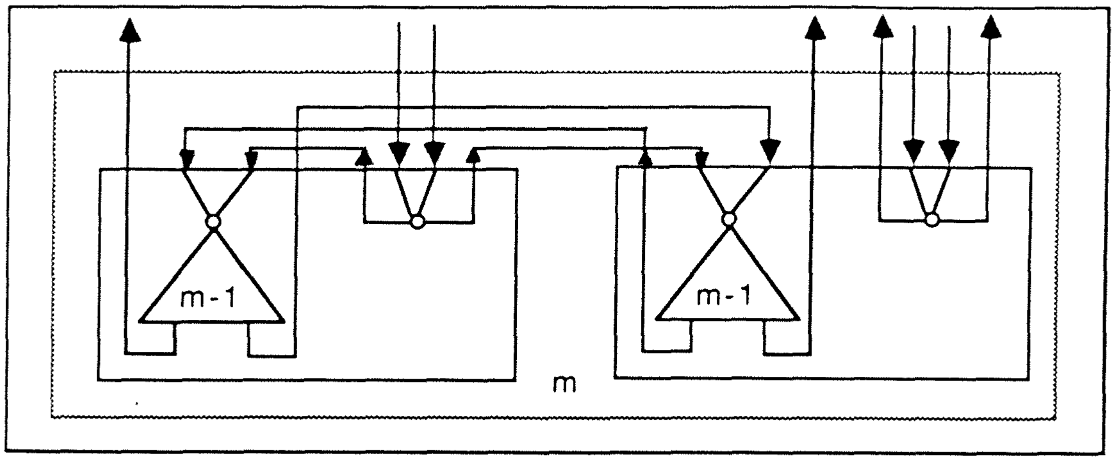

Figure 7.9. Extension of the Two (m-1)-level Sneptrees to an m-level Sneptree

Let one chip consist of an (m–1)-level H-structure Sneptree with four dangling links and a single processor with its four links. Two of the four dangling links in the Sneptree are out of the leftmost and the rightmost leaf nodes, respectively. The other two are the incoming links to the root node. There are eight connectors in a single chip as shown in the solid box in Figure 7.9.

Figure 7.9 illustrates how to connect two such chips into one m-level Sneptree. The resulting layout contains one m-level Sneptree with four dangling links and a single processor, the layout now being able to extend to bigger structures recursively.

7.5 Summary

The Sneptree is a versatile interconnection network for distributed computation. The boundary problem of a binary tree is eliminated in the Sneptree so that the mapping of an over-sized computation tree is done automatically. Moreover, a complete binary tree of arbitrary size can be mapped onto a Sneptree optimally. And a left/right skewed tree can be mapped onto a Cyclic Sneptree optimally.

The Sneptree is also suitable for VLSI implementation. It is possible to build a Sneptree of any size in a single chip with an area proportional to the total number of processors. The H-structure layout of the Sneptree is regular and can be constructed recursively. The number of crossings due to the extra links is proportional to the number of nodes of the Sneptree. The longest wire length is about the same as that in an H-structure binary tree. Furthermore, the Sneptree can be expanded easily by connecting two or more chips together.

7.5.1 Comparison with other related works

Like the Sneptree, the X-tree [15] is an augmented binary tree with identical nodes. Three ports per node, four ports per node and five ports per node are considered. The degree of each node is not fixed but the maximal degree is limited by the number of ports per node. Besides the binary tree connection, the extra ports can be connected arbitrarily. The main purpose of the X-tree is to provide fault-tolerance and uniform message traffic.

The Hypertree [18] is a binary tree with extra horizontal links (i.e., the links connecting the nodes located at the same level). The horizontal links provide a set of n-cube connections. Four ports per node and five ports per node are considered. Similarly, the main concern of the Hypertree is to provide fault-tolerance and shorten the distance between two arbitrary leaf nodes.

De Bruijn Networks [42] are a class of fixed degree logarithmic networks with an arbitrary number of nodes and degree. A De Bruijn Network with $(2^n - 1)$ nodes and degree 4 happens to be a Sneptree (Figure 7.10). Such networks are good for a communication network since the optimal routing path can be decided with local information and fault-tolerance is easily provided.

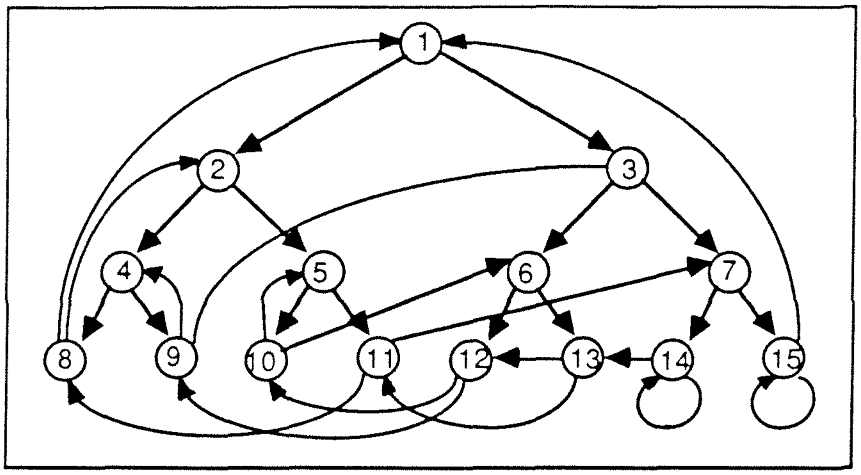

Figure 7.10. A de Bruijn Network with 15 Nodes and Degree 4

Compared with the other similar networks, the Sneptree is the only network which can simulate an oversized binary tree. The X-tree and the Hypertree contain extra links between sibling nodes so that it can simulate ring connection or n-cube connection. They cannot handle the mapping of an oversized problem well. The de Bruijn network with degree 4 is one type of Sneptree. The connection pattern is neither cyclic, symmetric nor regular. There is no regular way to lay out or extend the network.

7.5.2 Different Connection Patterns

From Theorem 7.1, we know there are many different connection patterns for the Cyclic Sneptree. It is interesting to compare the performance of different connection patterns of the Cyclic Sneptree in terms of the communication distance and the mapping performance of an unbalanced tree.

Figure 7.11.a shows another Cyclic Sneptree [33]. The numbers attached to the nodes show the node ordering in the left spanning cycle. Symmetrically, the right spanning cycles can be represented by node sequence (1,5,4,3,2,7,6,1). Such a connection pattern also has a regular structure and hence can be generated recursively. It is interesting to observe that this connection is planar if we switch the position of the leaf node pairs (3,7) and (6,4) as shown in Figure 7.11.b. Comparing the Cyclic Sneptree shown in Figure 7.2 with this one, the latter one (Figure 7.11.b) contains four duplicate links, i.e., (2,3), (4,5), (3,4)

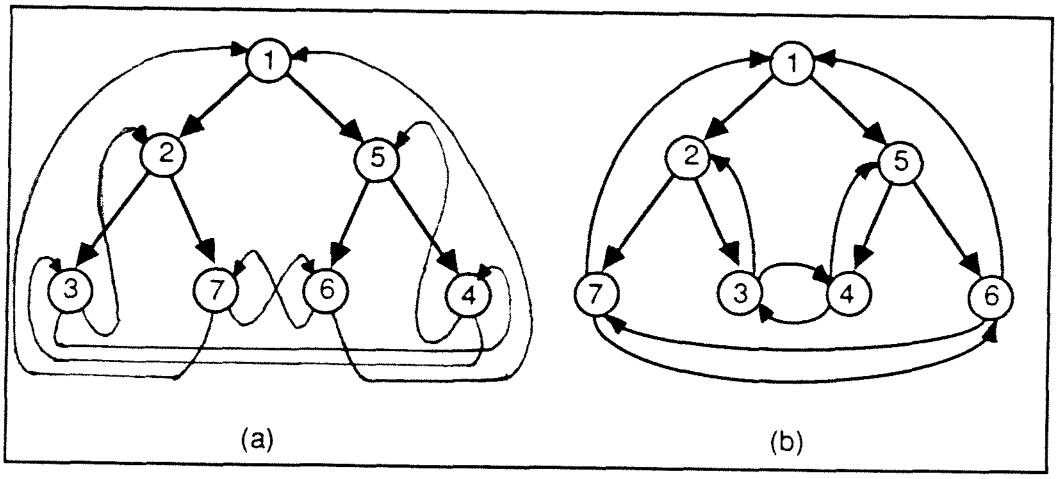

Figure 7.11. Another Cyclic Sneptree

and (6,7) while the former one (Figure 7.2) has only two duplicate links, i.e., (3,4) and (6,7). The duplicate links prevent the Sneptree from connecting more nodes together. Hence, the second connection pattern doesn't perform as well as the first one in terms of communication.

There are many other connection patterns for the Cyclic Sneptree; some of them may perform better than the one we chose in terms of the communication distance between two arbitrary nodes and the mapping performance of an unbalanced tree. But only the two connection patterns discussed above can be constructed recursively from the smaller Sneptrees without breaking the internal Sneplinks in the smaller structures. This property is important for VLSI implementation.

Despite the optimal mapping for a skewed tree, the Cyclic Sneptree is not necessarily the best for mapping an arbitrary unbalanced binary tree. A connection pattern which has the outgoing links of the leaf nodes in one half of the tree directed to the incoming links of the other half of the tree is probably the best choice for mapping an unbalanced binary tree, because the Sneplinks provide full connections between the two halves of the Sneptree such that mapping a tree node onto a leaf node of the Sneptree causes the subtree to be mapped onto the other half of the Sneptree. We call this type of connection the *Exchange Sneptree.*

Definition: An *Exchange Sneptree* is a Sneptree in which the outgoing Sneplinks of the leaves in the left half of the Sneptree are directed to the incoming Sneplinks of the nodes in the right half of the Sneptree plus one incoming link of the root, and similarly for the other half of the Sneplinks.

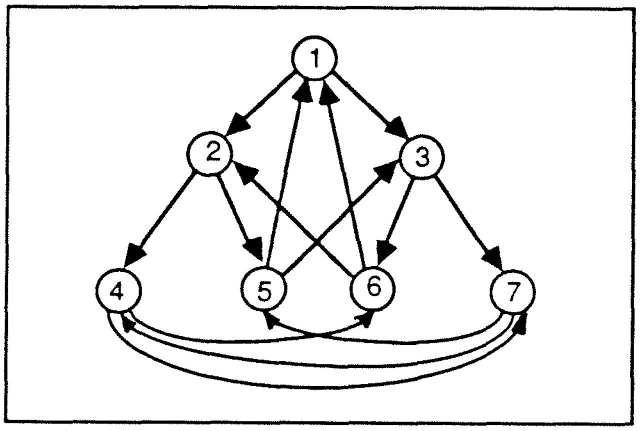

Figure 7.12. A three–level Exchange Sneptree

One example of the Exchange Sneptree suggested by C.F. Ho is shown in Figure 7.12. This connection is symmetric but neither extensible nor cyclic. More importantly, this connection has a very nice property: no matter which node the root is mapped onto, it results in a nearly optimal mapping for a complete binary tree. As we shall see in Chapter 9, this Exchange Sneptree is the best among seven different connection patterns that were chosen to map the process tree of the Sync Model.

Chapter 8

Routing In A Cyclic Sneptree

In this chapter, a leaf node routing algorithm for the Cyclic Sneptree in Figure 7.2 is described. The design of the routing algorithm is motivated by attempting to utilize the Sneplinks in the Sneptree to shorten the routing distance between two leaf nodes. In the following, we call the Cyclic Sneptree in Figure 7.2 the "Sneptree."

8.1 Motivation

A Sneptree of height n can simulate a binary tree of any height. Therefore, a problem which can be solved on a binary tree of any size can be applied onto a fixed-sized Sneptree directly. We are also interested in mapping other data structures, such as linear arrays and n-dimensional matrices onto the Sneptree. Since there are two spanning cycles embedded in the Sneptree, an obvious mapping of a linear array is to map the linear array in prefix order, i.e., onto the left or the right spanning cycle in the Sneptree. Another possibility is to map the linear array onto the leaf nodes of the Sneptree sequentially. We call the first mapping *preorder mapping* and the second one *leaf mapping.*

The preorder mapping is equivalent to mapping a linear array onto a ring. The advantage is that there is a direct link between any two adjacent elements of the linear array, and all the nodes in the Sneptree are equally utilized so that maximum parallelism is achieved. In consideration of the computations involving non-adjacent elements in the linear array, it is important to observe that in a ring the communication time is proportional to the distance between the two elements. On the other hand, the distance between any two nodes in a binary tree is at most logarithmic in the size of the tree, which is much better than the linear distance in a ring. If we want to utilize the links other than those in the left spanning cycle to shorten the distance of two elements, we have to identify the location of the destination element, find a general routing algorithm to route the message, and prevent traffic congestion in the intermediate nodes. All three tasks stated above appear to be difficult.

The leaf mapping is an alternative to the preorder mapping. Although only half of the nodes in the Sneptree are used for the computation, this mapping has the following advantages:

- The naming of the leaf nodes matches the order of the elements in the linear array if the nodes of the Sneptree be ordered in a breadth-first normal order, with address 1 for the root node and addresses 2^{n-1} to $(2^n - 1)$ for the leaf nodes.

- To route a message from a leaf node to another leaf node is much simpler than between two arbitrary nodes. An O(n) routing algorithm will be presented in the next section.

The design of the routing algorithm is constrained by the following criteria: ***communication distance*** (to find a route as short as possible), ***congestion constraint*** (to use the extra links to avoid a traffic jam at the upper level nodes), and ***time constraint*** (to keep the routing time as low as possible).

The time to route a message from a node x to another node y is the sum of the message transmission time and the processing time at the source node and each intermediate node. Let t_p and t_c be the time of one processing step and the time of one message transmission between adjacent nodes respectively. Suppose $< x = x_0, x_1, \ldots, x_{k-1}, x_k = y >$ is the route which the message is sent through and $f(i)$ is the number of processing steps necessary to compute the following route at the intermediate node x_i. The total routing time is

$$\sum_{i=0}^{k-1} f(i)t_p + k \times t_c. \tag{8.1}$$

The algorithm presented next keeps both k and $f(i)$ as low as possible. In an n-level binary tree, it is clear that k is bounded by $2(n-1)$ and $f(i)$ is constant for all i so that the routing time is O(n). Notice that the bitwise operations, such as determining if one node is in the subtree of another node, is assumed to take constant time. In a Sneptree, it is obvious that the shortest distance of any two nodes is not longer than that in the binary tree due to the Sneplinks of the Sneptree. Hence, the second term, k, of Eq.(8.1) is smaller for the Sneptree than that for the binary tree. To keep the total routing time for the Sneptree in the same order of magnitude as for the binary tree, we have to keep $f(i)$ constant in the intermediate nodes. As a consequence, the algorithm can't always find the shortest route. It finds a route which is shorter and less congested than the one in a pure binary tree, in O(n) time.

A leaf node routing algorithm will be presented in the following subsection and the algorithm written in CSP-like notation is given in Appendix B.

8.2 The Routing Algorithm

Before describing the algorithm in detail, we first define the ***breadth-first normal ordering*** of the nodes and prove a couple of theorems.

Definition. The ***breadth-first normal ordering*** is an addressing method for a binary tree. The nodes in an n-level binary tree are numbered from 1 to $2^n - 1$. The root node has address 1, and the left descendant and the right descendant of a given node x have addresses $2x$ and $2x + 1$, respectively.

Suppose that each address is represented by an n-bit binary number; the addresses of the left and the right descendants of a nonleaf node are derived by shifting its address one bit to the left and adding 0 or 1 to it.

With such an addressing scheme, the binary address of the lowest common ancestor of any two leaf nodes can be easily decided, which is an n-bit binary number with leading zeros followed by the common prefix of the binary addresses of the two leaf nodes. Furthermore, the binary addresses of the left and the right corner leaf nodes of a subtree with height h have h trailing 0's and h trailing 1's, respectively.

In our routing algorithm, the source node computes and selects the shortest route between the source node and the destination node. Then, a four-variable message carries all the routing information necessary for a receiving node to determine the next node on the route. The intermediate nodes need not recompute the shortest routes.

Suppose a message is sent from a leaf node x to a leaf node y in a Sneptree of height n. Without loss of generality, we assume x<y; i.e., node x is to the left of y. Let A be the lowest common ancestor of x and y, B and E be two direct descendants of A, and triangles BCD and EFG be the two subtrees containing x and y (see Figure 8.1). In the sequel, UV denotes the shortest path between two nodes U and V, and |UV| represents the length of this path. Four possible routes between x and y are (xB,BAE,Ey), (xD,DE,Ey), (xB,BF,Fy) and (xD,DEABF,Fy). The lengths of these four routes are |xB| + |Ey|+2, |xD| + |Ey|+1, |xB| + |Fy|+1, and |xD| + |Fy|+4, respectively. In order to find the shortest route of the four candidates, we need to compute |xB| ,|Ey|, |xD| and |Fy|. Notice that |xB| and |Ey| are bounded by the height of the triangle BCD (or EFG), and |xD| and |Fy| are bounded by twice of the height of the minimal subtree containing x and D (or y and F). The routing algorithm takes advantage of the Sneplinks within the triangles to find the shortest paths xB,xD,yE, and yF.

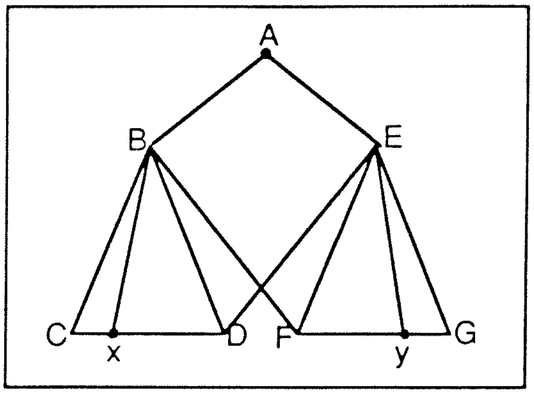

Figure 8.1. Four Possible Routes Between x and y

The length of xB (or yE) can be computed recursively. The shortest distance between B and a leaf node is 2 regardless of the height of B. The leaf nodes which have distance 2 from B are the two inner corner leaf nodes of the two subtrees of node B, and the shortest paths take one of the treelinks to a descendant of B and then take the Sneplink to the leaf (see Figure 8.2.a). Let a and b be the two direct descendants of B and let c and d be the two corner leaf nodes which have distance 2 from B. Then the leaf nodes which are at distance 3 from B are the nodes at distance 2 from nodes a or b, as well as the nodes at distance 1 from nodes c or d. There are six such nodes: two (a1 and a2 in Figure 8.2.b) are at distance 2 from a, two (b1 and b2) at distance 2 from b, and c1 and d1 are at distance 1 from c and d, respectively. Applying this technique recursively, we can find the shortest path between any leaf node and node B. The shortest path xB for an arbitrary leaf node x is a path starting from node B, following the treelinks down to a certain level of the tree, then taking the Sneplink to a leaf node and following the shortest route from this leaf node to node x (see Figure 8.3.a).

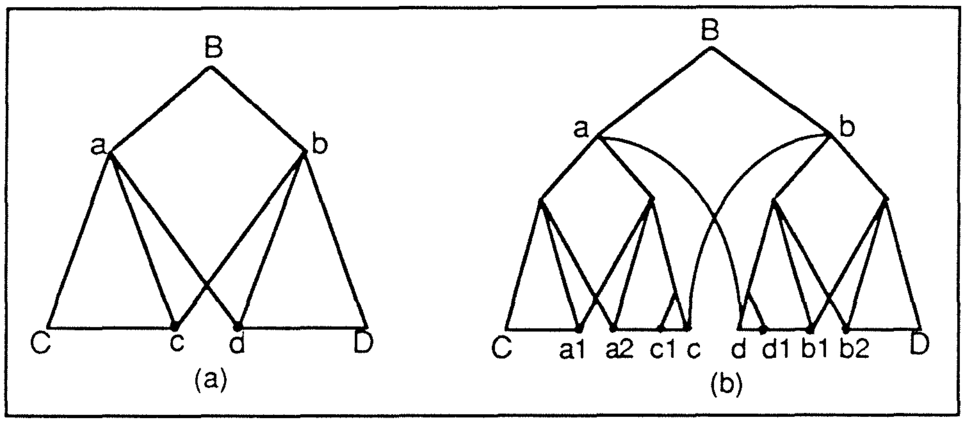

Figure 8.2. The Leaf Nodes with Distance 2 or 3 from Node B

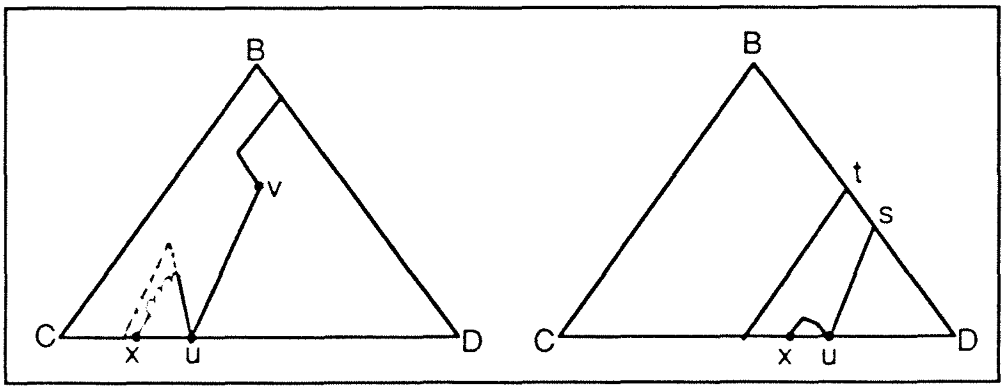

Figure 8.3. The Shortest Paths xB and xD

The program in Figure 8.4 is the first part of the routing algorithm which finds the shortest path among four candidates. The program is written in CSP-like notation and is performed in the source node x. To find xB, the algorithm finds a sequence of leaf nodes whose binary addresses differ from x only in trailing bits, and their trailing bits are all 1's or all 0's. Those leaf nodes can be routed to node B through a Sneplink so that the distance to B is shorter that the height of B. The length of a route from x via one of such leaf node, say x_j, to B is the sum of the shortest distance of x to x_j and the distance of x_j to B. After all such routes have been computed, the shortest one is the candidate to the shortest route of xB. If the shortest distance is longer than the height of B, then the direct route from x to B (going upwards through treelinks) is the shortest route. For instance, let the binary address of x be 00111010, where we ignore the leading bits that are the common prefix of the binary address of x and y because they are irrelevant in computing xB. The height of B is 7 so that the shortest distance of x and B won't exceed 7. The first leaf node which can take advantage of one of the Sneplinks is x_1=00111011 (x_1 is derived by changing the LSB of x to 1), which is of distance 1 from x and 6 from B by taking the Sneplink. Hence, the distance is 7 by routing through x_1. The second leaf node is x_2=00111111, which happens to be node c in Figure 8.2.a and has distance 2 from B. The distance of x and x_2 can be computed recursively and it turns out to be 4. Therefore, the distance of xB by routing through x_2 is 6, which is shorter than 7. Since there are no other leaf nodes which can take advantage of the Sneplinks, we can conclude that the shortest route of xB is from x to x_2, taking the Sneplink to the right descendant of B, and then up to B.The length of the shortest route is 6.

The distance of xD can be derived during the computation of xB. Let the lowest common ancestor of x and D be t, the right descendant of t be s, and the leaf node at the other end of the Sneplink out of s be u. Then, the route (Bs,su,ux) is one of the candidates for the shortest path Bx. Hence, the distance of ux is computed while computing xB and the shortest path between x and D is (Ds,su,ux) whose distance can be derived immediately (see Figure 8.3.b).

In the computation described above, we need to find the shortest route from x to another leaf which is a corner node of a subtree containing x. Again, this distance can be computed recursively. For instance, to route xu in Figure 8.3.a, we are trying to find a sequence of leaf nodes starting with x and ending with u. Each pair of adjacent nodes in the sequence has Hamming distance 1. We now route the message through the nodes in the sequence. The distance from x to any intermediate node can be computed recursively by the previous value. Let u be a right corner leaf node of a subtree and let the node sequence

1. Program at node $x = (a[n], a[n-1], \ldots, a[1])$

```
% The first part of this program computes the shortest distance among four
% candidates. Let d1,d2,d3 and d4 be the length of xB, sD, yE and yF in
% Figure 8.1, and d be the shortest one of the four.

i:=n;
% find the common ancestor of x and y,
{a[i]=b[i] -> i:=i-1};  % i is the height of their common ancestor

% from now on, the computation is based on the relative addresses of x and y,
% i.e., the addresses (a[i],a[i-1],...,a[1]) and (b[i],b[i-1],...,b[1]).
g1:=i-1; k:=i;      % g1: the height of B
{a[i]=1 -> i:=i-1}; g2:=i; % g2: the most significant 0 bit of x
{b[k]=0 -> k:=k-1}; g3:=k; % g3: the most significant 1 bit of y

% find the shortest distance of xB (d1) and xD (d2)
% w1 and w2 are the shortest distance of x to leaf nodes with address
% (a[i],..., a[j],0,...,0) and (a[i],..., a[j],1,...,1), respectively, if a[j-1]=0,
% and reverse  if a[j-1]=1, where j≥3.
[ a[2]≠a[1] -> w1,w2:=1,2
| a[2]=a[1] -> w1,w2:=0,3];

[g2=0 -> d2:=0
|g2=1 -> d2:=1];

% k is the distance from B to the corner leaf node under inspection,
% d1 is initially set to the height of B, and later set to the smaller one of the
% current route and previous minimal value.
% Also decide the values of m1,h1,l1 and dir, which is important routing information.
%
j:=3; k:=g1-j+2; d1:=g1;
m1:=0; h1:=g1; l1:=0, dir:=0;
t1:=0; t2:=0;
{j<=g2+1 -> l1:=t2; d2:=w2;
           [a[j]≠a[j-1] -> [w1+k<d1 -> d1:=w1+k; h1:=k-1; m1:=t1;
                                        [a[j-1]=1 -> dir=-1
                                        |a[j-1]=0 -> dir=1]
                           |w1+k≥d1 -> skip];
                           [w1+j≤2*j -> w1,w2:=w2,w1+j; t1,t2:=t2,t1
                           |w1+j>2*j -> w1,w2:=w2,2*j; t1.t2:=t2,j]
           |a[j]=a[j-1] ->  [w2+j≤2*j -> w2:=w2+j
                             |w2+j>2*j -> w2:=2*j; t2:=j]
           ];
           j:=j+1; k:=k-1
};
```

Figure 8.4. The Program to Compute the Shortest Route

```
% Find the shortest distance of yE (d3) and yF (d4) and set m2,h2 and l2.
[ b[2]≠b[1] -> w1,w2:=1,2
| b[2]=b[1] -> w1,w2:=0,3];
[g3=0 -> d4:=0
|g3=1 -> d4:=1];

j:=3; k:=g1-j+2; d3:=g1;
m2:=0; h2:=g1; l2:=0;
t1:=0; t2:=0;
{j<=g3+1 -> l2:=t2; d4:=w2;
            [b[j]≠b[j-1] -> [w1+k<d3 -> d3:=w1+k; h2:=k-1; m2:=t1;
                             |w1+k≥d3 -> skip];
                             [w1+j≤2*j -> w1,w2:=w2,w1+j; t1,t2:=t2,t1
                             |w1+j>2*j -> w1,w2:=w2,2*j; t1.t2:=t2,j]
            |b[j]=b[j-1] ->  [w2+j≤2*j -> w2:=w2+j
                              |w2+j>2*j -> w2:=2*j; t2:=j]
            ];
            j:=j+1; k:=k-1
};

% d: the shortest distance of x and y
d:=min(d1+d3+2,d2+d3+1,d4+d1+1,d3+d4+4);
```

Figure 8.4. The Program to Compute the Shortest Route (cont.)

be $(x = x_0, x_1, \ldots, x_k = u)$, where x_i is derived from x_{i-1} by changing the least significant 0 of x_{i-1} to 1, and k is the Hamming distance of x and u. Notice that the address of u has m trailing 1's, where m is the height of the minimal subtree containing x and u. The recurrence relation is

$$|x_0x_i| = \min(|x_0x_{i-1}| + j,\ \ 2 \times j) \tag{8.2}$$

where j is the position of the bit that differs in x_{i-1} and x_i. All the bits to the right of the j-th bit of x_{i-1} and x_i are 1's (Figure 8.5). The shortest route from x_{i-1} to x_i takes the left Sneplink of x_{i-1} to an ancestor node of x_i and then takes the right tree links down to x_i. The distance is j, i.e., the height of the lowest common ancestor of x_{i-1} and x_i. Furthermore, the shortest route between x_0 and x_i could be either the route $(x_0x_{i-1}, x_{i-1}x_i)$ or the route containing only treelinks and passing through the lowest common ancestor of x_0 and x_i (the distance of this route is $2 \times j$). In the previous example, the shortest route from x=00111010 to x_2=00111111 is taking the Sneplink to leaf node x_1=00111011, then Sneplink again to the ancestor of x_2 and taking the right treelinks down to x_2. The distance is 1(x to x_1)+3(x_1x_2)=4.

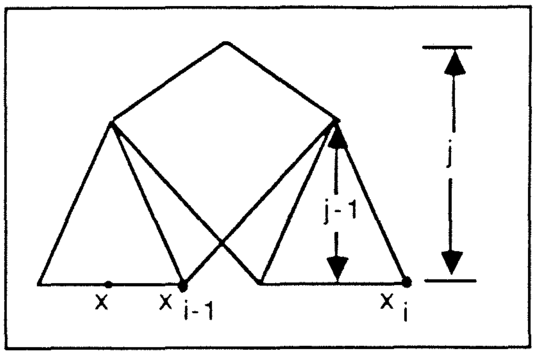

Figure 8.5. The Shortest Path Between x_{i-1} and x_i

Similarly, if u is a left corner leaf node (with trailing 0's), |xu| can be derived by computing the distance of x and a series of intermediate nodes whose addresses are derived by changing the least significant nonzero bits of x to 0 until it reaches u.

Now we can select the shortest path among the four candidates, (xB,BAE,Ey), (xD, DE,Ey), (xB,BF,Fy) and (xD,DEABF,Fy). So far, all the computation and decision being made are accomplished at the source node x. To achieve O(n) time performance, we don't want to repeat the computation in any other intermediate nodes along the route. Hence, the routing information should be sent to the intermediate nodes to guide them to select the proper next node along the route. It appears that a four-variable message is enough to carry the route information and avoid extra computation.

When the shortest route goes through xB, two figures, $m1$ and $h1$, are needed to guide the route. Suppose the message is routed from x to another leaf node u, then taken the Sneplink to a nonleaf node v and sent up to node B (Figure 8.3.a). We first need to know the route information of xu. If the route xu follows the treelinks up and down to some intermediate leaf node (i.e., when $2j$ is the smaller one in Eq.(8.2)). we record the highest point of this route in variable $m1$; otherwise, $m1$ is zero. The second figure $h1$ is the distance of B and v. When xB contains treelinks only, $m1$ is zero and $h1$ is the height of node B. Similarly, $m2$ and $h2$ provide the corresponding information needed to describe the route yE. Furthermore, we need a direction flag, dir, to guide the message to either the left descendant ($dir = 1$), the right descendant ($dir = -1$) or the father node ($dir = 0$).

When route xD is in the shortest path, the only information ($l1$) we need to know is the highest point that the route reaches through the treelinks. (i.e., when $2j$ is the smaller one in Eq.(8.2).) When the route contains no upward treelinks, $l1$ becomes zero. Similarly, $l2$ is the corresponding information needed to describe route yF.

Let a four-variable message be $(level1, dir, level2, dest)$. In general, the routing information for the route in triangle BCD is carried in the first two variables of the message. The third variable carries the information for the route in triangle EFG. The last variable is always the destination. More specifically, $level1$ carries the value of $l1$ or $m1$ depending on which route is selected. Variable dir is usually a three-value variable used to select the next node in the route when a leaf node or the highest nonleaf node is reached. When xB is selected, $h1$ is also carried by dir. Variable $level2$ carries either the value of $h2$ or $l2$ depending on whether yE or yF is selected in triangle EFG. The value of $m2$ has to be reproduced by a specific node in route Ey when Ey is selected, since there is no room to carry the value in a message. That specific node is the lowest nonleaf node traveling down from E through treelinks, from where the route takes the Sneplink to a leaf node and then takes the shortest route to y. Such a node corresponds to node v in route xB (see Figure 8.3.b). The second variable is also used to select one of the four possible routes. When dir is used to carry the direction information of the route in the triangle BCD, the route information for node A,B and E is carried by the fourth variable instead. For instance, the value of the fourth variable is negative when yF is selected. The routing information for triangle EFG will be resumed at node A,B or E by moving the information carried by the third and the fourth variables back to the first two variables.

The routing program is quite complex because each node makes decisions based on the origin of the message and the values of the message. Figure 8.6 shows the programs for the second part of the routing algorithm. Three different programs are needed for the source node x, any nonleaf node, and any leaf node other than x, respectively. Different messages are sent to different ports based on the routing information received or computed. For more details, see the comments in the program.

1. **Program at node** $x = (a[n], a[n-1], \ldots, a[1])$ (Cont. from Figure 8.4)

```
% send proper routing messages to one of its four neighbors according to the
% shortest route, the values of dir, m1, h1, l1, h2 and l2.
% R,L,F,S are the four ports connected to the left descendant, the right descendant,
% the father and the other ancestor linked by the Sneplink, respectively.
[d=d1+d3+2 -> [dir=1 -> [m1=0 -> R!(h1,1,h2,y)
                        |m1≠0 -> F!(m1,1+h1,h2,y) % carry m1,h1,dir and h2
                        ]
              |dir=-1 -> [m1=0 -> L!(h1,-1,h2,y)
                         |m1≠0 -> F!(m1,-1-h1,h2,y)
                         ]
              |dir=0 -> F!(h1,-1,h2,y) % carry h1 and h2
              ]
|d=d1+d4+1 -> [dir=1 -> [m1=0 -> R!(h1,1,l2,-y)
                        |m1≠0 -> F!(m1,1+h1,l2,-y)
                        ]
              |dir=-1 ->[m1=0 -> L!(h1,-1,l2,-y)
                        |m1≠0 -> F!(m1,-1-h1,l2,-y)
                        ]
              |dir=0 -> F!(h1-1,0,l2,y)
              ]
|d=d2+d3+1 -> [l1=0 -> L!(0,-1,h2,y)  % carry h2
              |l1≠0 -> F!(l1,-1,h2,y)
              ]
|d=d2+d4+4 -> [l1=0 -> L!(0,-1,l2,-y) % special routing info. in the fourth
              |l1≠0 -> F!(l1,-1,l2,-y) % variable
              ]
].
```

Figure 8.6. The Programs for Routing Messages

2. Program at an nonleaf node $t = (t[n], t[n-1], ..., t[1])$

```
% When the message comes from its left or right port and m>0, simply send the
% message to its father.  If m=0, the highest point along the route is reached
% and then send the message to one of the descendants or the Snep port according
% to the value of dir.
[L?(m,dir,h,y) -> [m≠0 -> F!(m-1,dir,h,y)
                  |m=0 -> [dir<-1 -> R!(-dir-1,-1,h,y)
                                     % at the height point along route xB
                          |dir=-1 -> i:=0; % at node A when (xB,BAE,Ey) is chosen
                                 t[n]=0 -> t:=t*2; i:=i+1; % i: the height of t
                                 R!(h,0,i-1,y) % recover h2
                          |dir= 0 -> S!(h,-1,0,y)
                          ]
                  ]
|R?(m,dir,h,y) -> [m>0 -> F!(m-1,dir,h,y)
                   |m=0 -> [dir≠0 -> L!(dir+1,-1,h,y)
                           |dir=0 -> S!(h,-1,0,y)
                           ]
                   |m<0 -> L!(-1,0,h,y) % at node A when (xD,DEABF,Fy) is chosen
                   ]
|F?(m,dir,h,y) -> [dir=-1 -> R!(m,-1,h,y)
                   |dir=1 -> L!(m,+1,h,y)
                 % when dir=m=0, it is a node along route yE and m2 and dir is
                 % recomputed here.
                   |dir=0 -> [m<0 -> S!(h,dir,0,y)
                                 % at node B when (xD,DEABF,Fy) is chosen
                             |m=0 -> [t[1]=1 -> dir:=1
                                     |t[1]=0 -> dir:=-1
                                     ]
                                     i:=0;
                                 % find the height of t (i)
                                     {t[n]=0 -> t:=2*t; i:=i+1};
                                     j:=1; d:=0; m2:=0;
                                 % recompute the shortest path between y and a leaf
                                 % node directed by the Sneplink from the current
                                 % node.
                                     {j<=i and b[j]≠ t[i+1] ->
                                              [d+j≤2*j -> d:=d+j
                                              |d+j>2*j -> d:=2*j; m2:=j];
                                              j:=j+1};
                                     S!(m2,dir,0,y)
                             |m=1∧h≠1 -> [b[h]=0 -> R!(0,0,0,y)
                                     |b[h]=1 -> L!(0,0,0,y)
                                      ]
                             |m>1∨h=1 -> [b[h]=0 -> L!(m-1,0,h-1,y)
                                     |b[h]=1 -> R!(m-1,0,h-1,y)
                                     ]
                             ]
                   ]
|S?(m,dir,h,y) -> i:=0;
```

```
        {t[n]=0 -> t:=2*t; i:=i+1}; % i: height of t
        j:=n;
        { j>i and t[j]=b[j] -> j:=j-1};
        [j<=i -> % y is in the subtree of t
                    [y<0 -> F!(-1,1,h,-y) %node E when (xD,DEABF,Fy) is chosen
                % going down the tree according to the address of y
                % when h=1, take opposite direction so that it can take
                % the Sneplink later to switch to the other subtree
                    |y>0 ->[ h=1 -> [b[j]=0 -> R!(0,0,j-1,y)
                                    |b[j]=1 -> L!(0,0,j-1,y)
                                    ]
                           | h>1 -> [b[j]=0 -> L!(h-1,0,j-1,y)
                                    |b[j]=1 -> R!(h-1,0,j-1,y)
                                    ]
                           ]
                    ]
        |j>i -> % j is the height of common ancestor of y and t
                    k:=j-i;  % the distance from t to B
                % along the route xB, if k=m (which carries h1), then
                % sending up, otherwise sending down.
                    [k=m -> [y>0 -> F!(m,-1,h,y)
                           |y<0 -> F!(m-1,0,h,-y) % recover y when (xB,
                                % BF,Fy) is chosen.
                           ]
                    |k>m -> [dir=-1 -> R!(m,dir,h,y)
                           |dir=1 -> L!(m,dir,h,y)
                           ]
                    ]
        ]

]
```

Figure 8.6. The Programs for Routing Messages (cont.)

3. Program at a leaf node t ($t[n], t[n-1], \ldots, t[1]$) other than the source node

```
% If the current node is the destination, stop.  Otherwise, if the message comes
% from either the father port or the Snep port, send the same message to its left
% or right port according to the value of dir.
% If the message comes from the left or the right port, then the current node is
% along the route yE or yF and the information m2 (for yE) or l2 (for yF) is carried
% by the first variable m.  The algorithm first finds the height of the common
% ancestor of y and t.  If the height (i) is equal to m, then the route climbs
% up i levels and downwards to reach y.  If the height is greater than m, then
% the route climbs up i-1 levels with the value m carried in the third variable.
[F,S?(m,dir,h,y) -> [y=t -> skip
                    |y≠t -> [dir=1 -> R!(m,dir,h,y)
                            |dir=-1 -> L!(m,dir,h,y)
                            ]
                    ]
|L,R?(m,dir,h,y) -> [y=t -> skip
                     |y≠t -> i:=h;
                             {y[i]=t[i] -> i:=i-1};
                             [i=1 -> S!(0,dir,h,y)
                             |i=m -> F!(i,dir,h,y)
                             |i>m -> F!(i-1,0,m,y)
                             ]
                     ]
]
```

Figure 8.6. The Programs for Routing Messages (cont.)

8.3 Performance

The computation time in the source node x is O(n). When Ey is selected, one of the intermediate nodes along route Ey needs to reproduce the value of $m2$ in k steps, where k is the height of this specific node. When xB is selected, a few nonleaf nodes along the route need to compute the height of the lowest common ancestor of themselves and the destination node. Such a bitwise operation is again assumed to take constant time. In conclusion, only the source node and at most one intermediate node need to do some computation in O(n) time. From Eq.(8.1), we can conclude that the routing algorithm takes O(n) time to route the message from the source to the destination.

The result of the routing algorithm gives a good approximation to the shortest path of xy. Furthermore, the routing algorithm always finds the shortest path within the triangle ACG. This routing algorithm uses only the links local to the minimal subtree containing the source and the destination nodes. The Sneplinks external to this subtree are never considered. As a consequence, the two Sneplinks of the root node are never used for routing. Because of this restriction, the routing algorithm does not always compute the shortest path. (For example, the route from the left corner leaf to the right corner leaf has a distance 2, whereas our algorithm chooses a route of length twice the height of the tree.) However, this restriction has many advantages. The algorithm is simple and yet computes nearly optimal routes, and the traffic of the upper level nodes is reduced.

In a binary tree, the nodes at the upper levels are the most congested nodes because half of the leaf nodes have to route through the root node to communicate with the other half of the leaf nodes. In case any leaf node is communicating with all the other leaf nodes, the root node has to transmit about half of the messages and the nodes one level down the root have to transmit 5/8 of the messages. Then, the traffic at each node decreases level by level from 13/32, to 29/128,.... In a Sneptree, four routes may be chosen to route a message between two arbitrary leaf nodes, and only two of them pass through the lowest common ancestor of the two leaf nodes. Assume the four alternatives are equally probable; then the traffic at the common ancestor is reduced to a half of the binary tree case and the traffic at the nodes one level down the common ancestor are reduced to three quarters since three of the four routes pass through that node (see Figure 8.1). In case any leaf node is communicating with all the other leaf nodes, the traffic at the top level nodes becomes 1/4, 7/16, 19/64, 43/256,... of the total amount of messages. The figures show that the traffic at the top level nodes is reduced to about half of the binary tree case. The actual figures depend on the height of the Sneptree. The traffic at the nodes of the same height is no longer the same. The exact figures need more analysis.

The simulation results show that the average routing distance of any two leaf nodes is getting closer to the optimal average distance when the Sneptree is bigger. The simulation also shows that for some specific communication patterns, the routing result is almost optimal, such as shift by 2^k operations, i.e. routing one leaf to another leaf at 2^k distance apart. Figure 8.7 shows the optimal results and our routing algorithm results for the average distance of any two leaf nodes and the average distance of a perfect shuffle operation. Figure 8.8 shows the average distance of shift by 2^k operations, in which the curves for routing results and optimal results are overlapped.

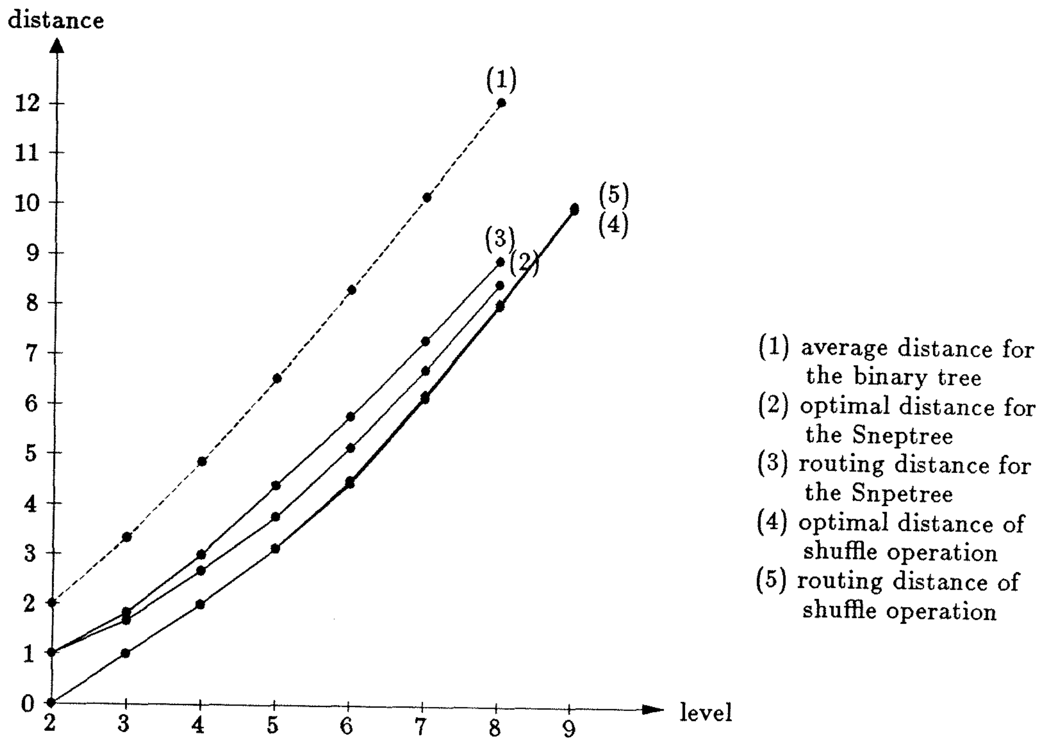

Figure 8.7. The Average Routing Distances

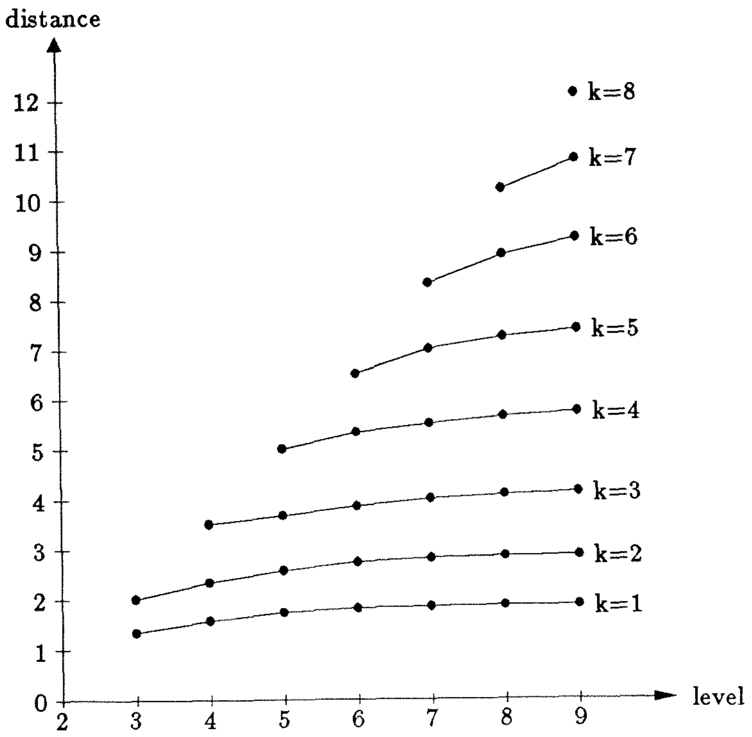

Figure 8.8. The Average Routing Distances of Shifting by 2^k Operations

Chapter 9

Mapping Of the Sync Model onto the Sneptree

The Sync Model creates a dynamic tree of processes. The root is an AND process for a single goal, and the OR processes and the AND processes alternate in any path from the root to a leaf. Communication channels also exist between sibling AND processes. The graph is no longer a tree, but we still call it a "process tree." The processes are created and destroyed dynamically during computation. To map such a process tree onto a Sneptree, we first map the process tree onto an unbounded binary tree, then map the binary tree onto the Sneptree. The second part of the mapping is done automatically by the connection of the Sneptree; therefore, we only need to worry about how to map a process tree onto an unbounded binary tree.

In the next section, we propose two mapping criteria, load-balance factor and communication overhead, for the performance evaluation of a mapping algorithm. In Section 9.2, a mapping algorithm to map the process tree onto a Sneptree is described. In Section 9.3, the performance of the mapping algorithm is analyzed in terms of the mapping criteria defined in Section 9.2. Some other mapping algorithms are compared. Different connection patterns of the Sneptree are also compared. In the last section, we summarize the contents of this chapter.

9.1 Mapping Criteria

Mapping problems (also referred as Graph Embedding Problems in the literature [22, 40, 56]) have appeared in many applications, such as representing data structures in computer memory, laying out circuits on chips, embedding distributed computations into a network of processors. In mapping a computation graph onto an implementation graph, the size of the computation graph may be greater than that of the implementation graph. Hence, one-to-one mapping is not always possible and we need to consider two factors in the measurement of the mapping performance.

1. Load Balance Factor:

   Balanced load in each node is the first criterion for a good mapping. The number of cells mapped in each node is used to measure the load of each node. Let the ***load factor*** be the number of cells assigned in each node and ***load balance factor*** be the standard deviation of the load factors. Suppose f_a is the load factor of node a, then the load balance factor, LF, of a mapping can be formulated as follows:

$$LF = \frac{\sqrt{\sum_a \left(f_a - \overline{f}\right)^2}}{N}$$

   where N is the size of the Sneptree and $\overline{f}$ is the average load factor; i.e., $\overline{f} = (\sum_a f_a)/N$. A good mapping is a mapping with a minimal load balance factor.

2. Communication Overhead:

   Minimal communication overhead is the second criterion for a good mapping. Since an edge in the computation graph is mapped to a path in the implementation graph, the communication overhead is measured in terms of the length increase of the communication paths for adjacent cells in the computation graph. The average communication overhead CO is defined as follows:

$$CO = \frac{\sum_e(|l_e| - 1)}{M - 1}$$

   where l_e is the path in the implementation graph to which an edge e in the computation graph is mapped and $|l_e|$ is its length. $(M - 1)$ is the total number of links in a process tree of size M.

9.2 Mapping Algorithm

As mentioned in the beginning of this chapter, we don't map a process tree onto the Sneptree directly. Instead, we first map the process tree onto a binary tree and then map the binary tree onto the Sneptree. A process tree is an arbitrary fanout tree with channels between sibling nodes at alternative levels. We adopt Browning's method [5] to map such an arbitrary tree onto a binary tree. The mapping algorithm is described as follows:

Mapping From a Process Tree onto a Binary Tree

Mapping Algorithm:

1. The root of the process tree is mapped onto the root of the binary tree.
2. If a cell a in the process tree, which has k descendants, is assigned to a node b, the k descendants are mapped onto the k leaf nodes of a height-balanced subtree rooted at node b. The mapping is constructed recursively as follows:
    a. If k=1, map the only descendant of a to the left descendant of b.
    b. If k≥2, map the first $\lceil k/2 \rceil$ descendants of a to the left descendant of b, and map the other $\lfloor k/2 \rfloor$ descendants of a to the right descendant of b.
3. When k cells are to be mapped to a node b,
    a. If k=1, map the only cell to node b and go to step (2).
    b. If k≥2, map the first $\lceil k/2 \rceil$ cells to the left descendant of b, and map the other $\lfloor k/2 \rfloor$ cells to the right descendant of b, and then go to (3).

Figure 9.1 illustrates the mapping of a process tree of height one and fanout five onto a binary tree of height three. The leaves of the process tree are mapped one-to-one onto the leaf nodes of the binary tree.

Notice that the binary tree is height-balanced, because the descendants of a given cell are evenly mapped onto the left and the right subtrees of the mapped node. The distances from a leaf node to the root differ by at most 1 in the binary tree. Also notice that in Figure 9.1 , a six-cell process tree is mapped onto a nine node binary tree, where three intermediate nodes (the nodes that are not a root or a leaf node) in the binary tree are

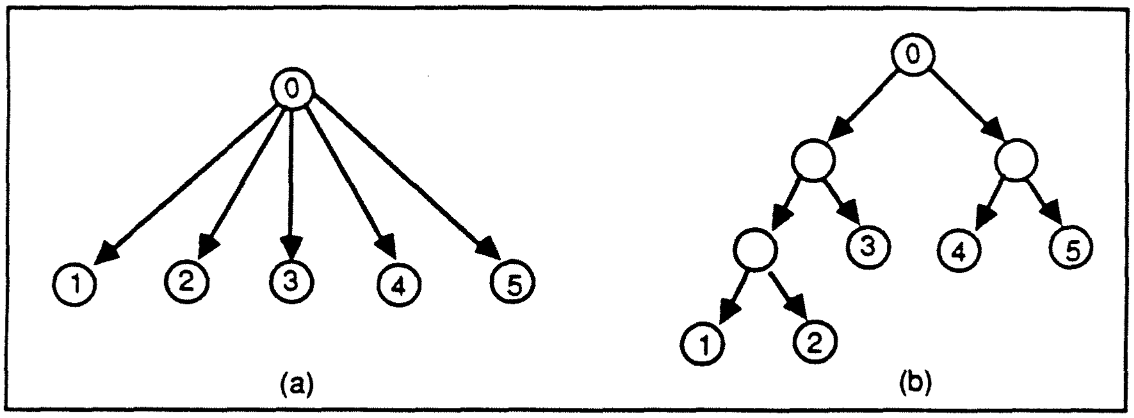

Figure 9.1. Mapping Example

not mapped by any cell. Such nodes are used as communication channels and thus given a name, the ***communication nodes.*** The channel between two sibling cells, say a and b, in the process tree is mapped onto a path in the binary tree starting from node a', up to the lowest common ancestor of a' and b', and then down to node b', where a' and b' are the nodes in the binary tree onto which cells a and b are mapped. The length of such a path is bound to $2\lceil \log_2 k \rceil$, where k is the fanout of the father of a and b.

This mapping algorithm has the following advantages:

1. One-to-one mapping:

   This mapping algorithm gives a one-to-one mapping from the process tree to a binary tree. Suppose the binary tree is always large enough to contain the process tree; the load factor of each node is either 1 or 0 so that the mapping is load-balanced. If we map the process tree in Figure 9.1.a onto a smallest binary tree such as in Figure 9.2 to eliminate extra communication nodes, we face problems while further extending the mapping. We can either map the descendants of node 1 onto nodes 2 and 3 again to keep node 1 and its descendants close or map all the descendants of nodes 1 to 5 onto unused nodes. The first alternative is no longer a one-to-one mapping; therefore, the mapping is not load balanced. The second alternative results in unbounded communication overhead for an edge in the computation graph.

2. Low Communication Overhead:

   Any two adjacent cells in the process tree are kept close in the binary tree. Any edge of a process tree of height 1 and fanout k will be mapped onto a path of length

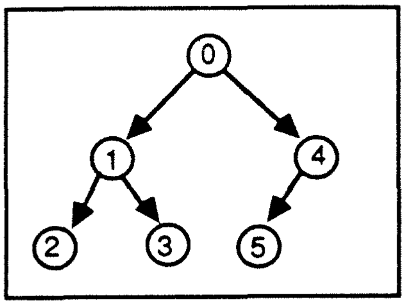

Figure 9.2. Smallest Binary Tree Mapping

$\lfloor \log_2 k \rfloor$ or $\lceil \log_2 k \rceil$ in the binary tree. The average communication overhead is about $(\log_2 k - 1)$. For a process tree with arbitrary height and arbitrary fanout, the average communication overhead can be approximated by the average of the summation of $(\log_2 k - 1)$ for all the cells with fanout $k > 2$ in the process tree. Compared with the minimal tree mapping in Figure 9.2, the communication overhead for a two-level process tree is higher in the former case. But if we do a one-to-one minimal tree mapping, i.e., we map the descendants of one cell (say, node 1) to the nearest unused nodes in the subtree of the node where the cell is mapped to, it is difficult to predict where the descendants of a cell will be mapped to and therefore hard to compute the communication overhead. The communication overhead is higher for a cell in the lower levels of the process tree. The mapping distance of two adjacent cells is dependent of the fanouts of all the predecessors in the process tree. Hence, there is no upper bound for the mapping distance of a given edge in an arbitrary process tree. In contrast, our mapping algorithm results in a minimal and fixed-length communication path for each pair of adjacent cells no matter where they are located in the process tree.

Mapping a Binary Tree onto a Sneptree

The mapping of an arbitrarily sized binary tree onto a Sneptree is done automatically. This mapping always maps adjacent nodes in the binary tree onto adjacent nodes in the Sneptree. In other words, one edge in the binary tree is mapped onto one edge in the Sneptree and therefore, there is no extra communication overhead introduced. In Chapter 7, we have proved that the mapping of a complete binary tree onto a Sneptree is optimal in terms of the load balance factor. We also presented a connection pattern, the Exchange Sneptree, which has the best connectivity between the two halves of the Sneptree so that it is good for mapping an unbalanced binary tree.

Integrating the two mapping algorithms, we are now able to map the process tree of our Sync model onto a fixed size Sneptree. In computing the load factor of the composite mapping, all the communication nodes in the binary tree are excluded because they are not mapped with any cell in the process tree. The communication overhead of the composite mapping is always less than that of the first part of the mapping because several nodes in the binary tree may be mapped onto one single node in the Sneptree, therefore, it may bring nodes in distance away together. The overall load and overhead are hard to analyze because it highly depends on the structure of the process tree, the connection pattern of the Sneptree, and the size of the Sneptree.

In the next section, we present the simulation results of our mapping algorithm for various types of connection patterns. We also compare the mapping performance of different mapping algorithms, which are the mapping algorithm described above integrated with different load balancing heuristics.

9.3 Mapping Performance

Two sample programs are chosen to run the mapping algorithm. One is *quicksort*, and the other one is a nondeterministic program *a* with both AND and OR parallelism. The two programs are described in Appendix B. We construct the complete process tree of each program and map it onto the Sneptree. Both programs are executed with different arguments. For instance, quicksort on 4, 5, 6 and 7 elements are simulated, similarly for *a*. Seven different connection patterns are considered, each with 15 nodes, 31 nodes or 63 nodes. The seven connection patterns are (1) Cyclic Sneptree (Figure 9.3); (2) Linear Cyclic Sneptree (Figure 9.4), where its left cycle is the same as (1), but its right cycle is in reverse direction as in (1); (3) De Bruijn Network (Figure 9.5); (4) Exchange Sneptree (Figure 9.6), where all the outgoing links of the leaf nodes at the left half are directed to the nodes in the right half and vice versa; (5) Self Looping Sneptree (Figure 9.7), where every leaf node has a loop back to itself; (6) Planar Cyclic Sneptree (Figure 9.8), a variation of Self Looping Sneptree by redirecting all the loops of the leaf nodes in the Self Looping Sneptree; and (7) Another Cyclic Sneptree (Figure 9.9), a variation of (2), where its right cycle is the same as (2), but its left cycle is in reverse direction as in (2).

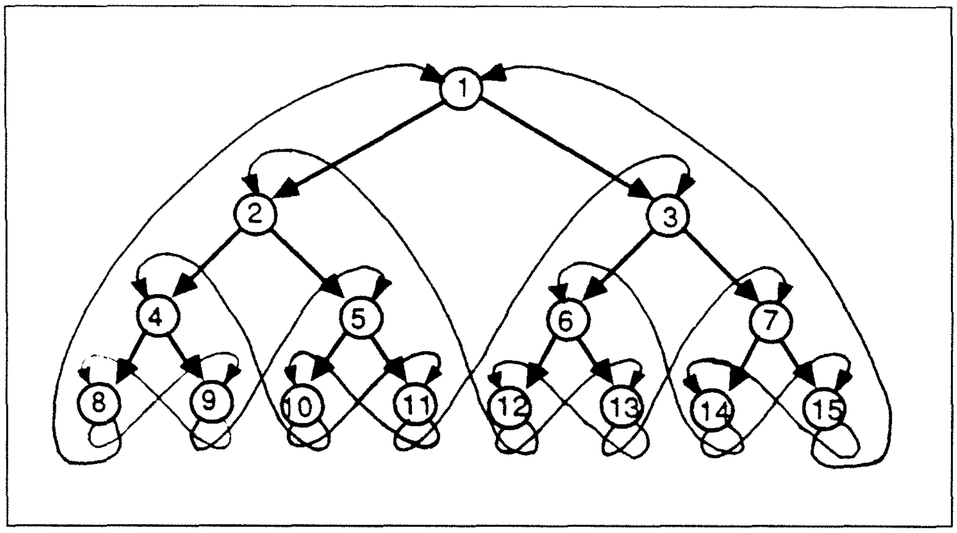

Figure 9.3. (1) 15-node Cyclic Sneptree

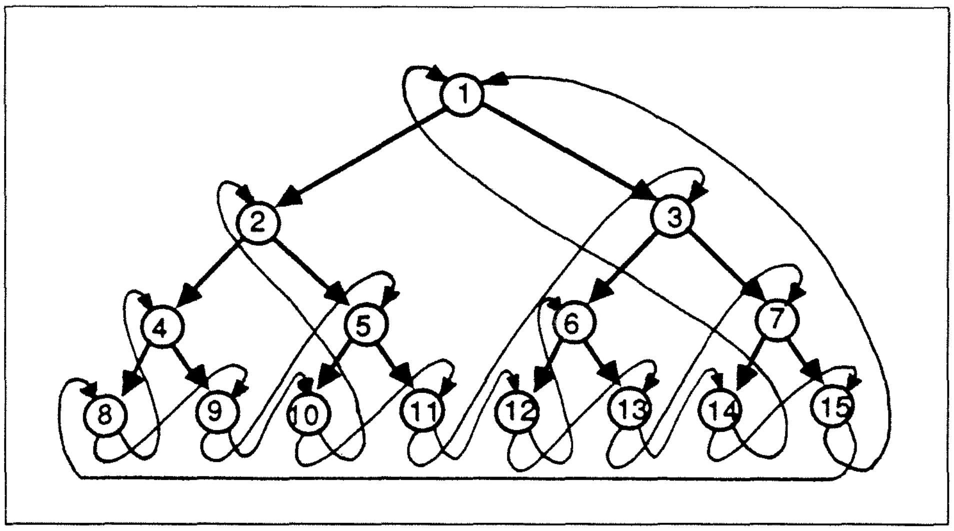

Figure 9.4. (2) 15-node Linear Cyclic Sneptree

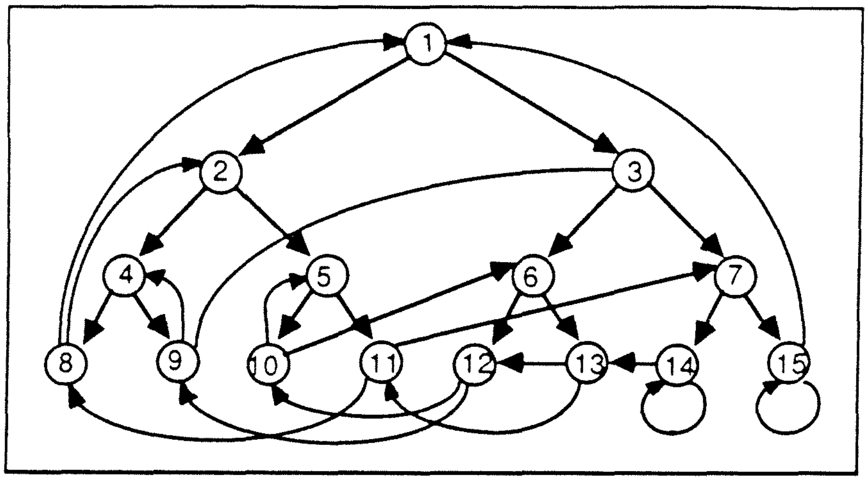

Figure 9.5. (3) 15-node De Bruijn Network

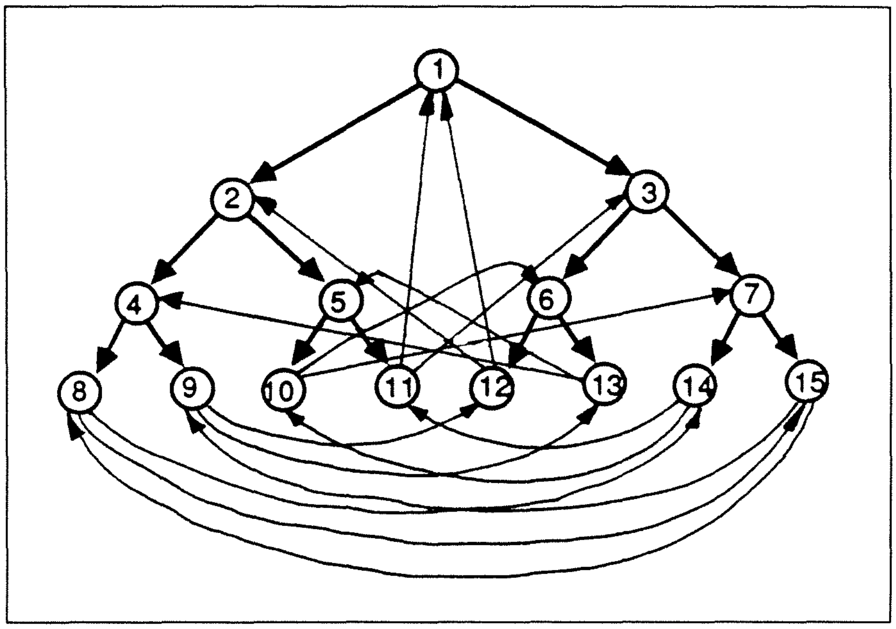

Figure 9.6. (4) 15-node Exchange Sneptree

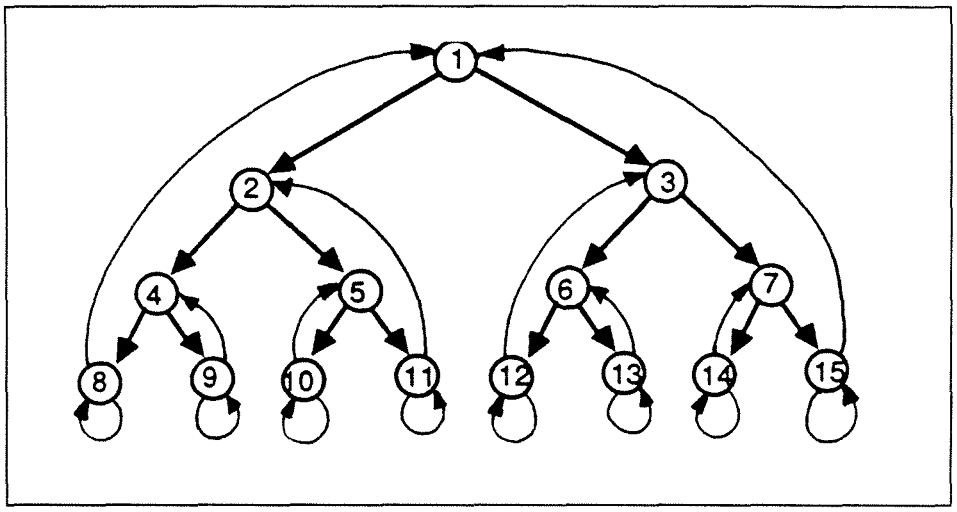

Figure 9.7. (5) 15-node Self-Looping Sneptree

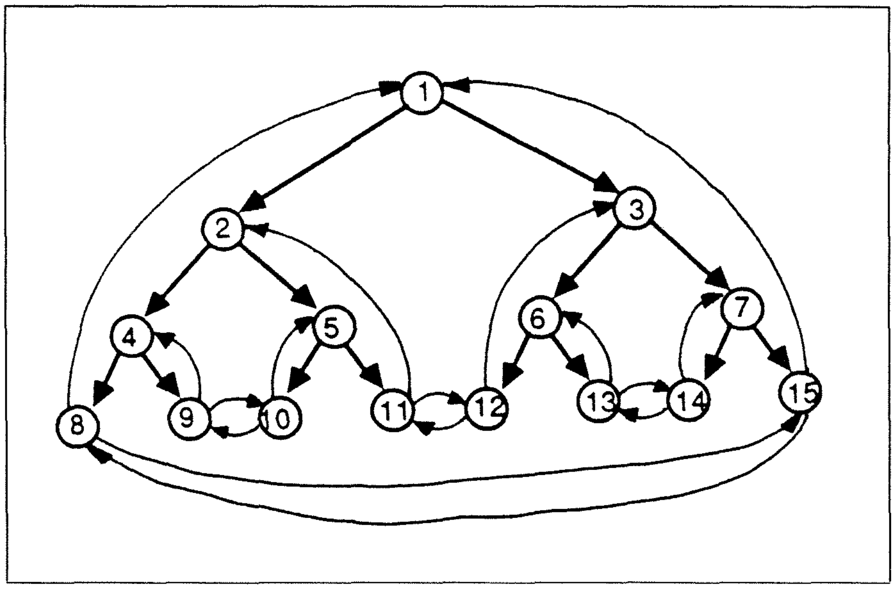

Figure 9.8. (6) 15-node Planar Cyclic Sneptree

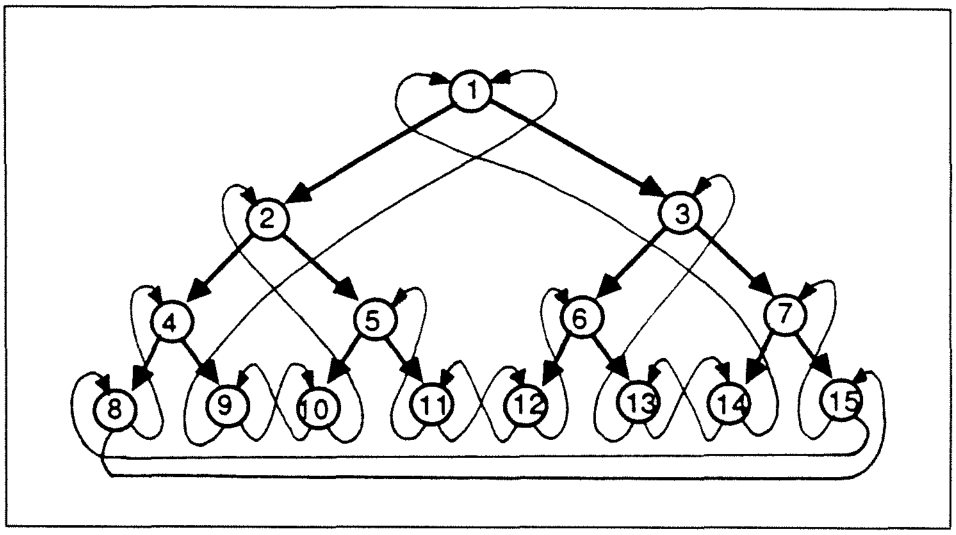

Figure 9.9. (7) 15-node Another Cyclic Sneptree

The properties of the seven connection patterns are compared in Table 9.10 in terms of their symmetry, extensibility, planarity and cyclicity.

Conn. Pattern	Cyclic	Symmetric	Extensible	Planar
Cyclic	Yes	Yes	Yes	No
Linear Cyclic	Yes	No	Yes	No
De Bruijn	No	No	No	No
Exchange	No	Yes	No	No
Self Looping	No	Yes	Yes	Yes
Planar Cyclic	Yes	Yes	Yes	Yes
Another Cyclic	Yes	Yes	Yes	No

Table 9.10. Comparison of Seven Different Connection Patterns

The following mapping algorithms are modified from the one described in Section 9.2 by adding different load-balancing heuristics. The heuristics is based on the load factors of adjacent nodes at a given node. Five Mapping Algorithms are attempted; they are:

V1. Same as the mapping algorithm described before except that, if a cell that has a single descendant, its descendant is mapped on the same node as the cell.

V2. Same as (V1), except that an OR process with a single descendant maps its descendant to the less loaded descendant of the node which the OR process is mapped onto.

V3. Same as (V2), except that an AND process with a single descendant also maps its descendant to the less loaded descendant of the node which the AND process is mapped onto.

V4. Define the *weight* of a mapping step as the number of cells to be mapped onto one particular node. This algorithm maps the set of cells with heavier weight onto the less loaded node in step (2.b) and (3.b) of the mapping algorithm described in Section 8.2. When a cell has only one descendant, it follows the mapping strategy in (V3).

V5. This algorithm is the reverse of (V4). It maps the lighter weighted set of cells onto the less loaded node in step (2.b) and (3.b).

We found that the simulation results of the five mapping algorithms vary for different programs, different size of the process trees, different size and different connection patterns of the Sneptrees, (see Appendix B, Figure B.5 to Figure B.10). Therefore, we conclude that there is no significant difference of the five chosen mapping algorithms for general applications.

Since we believe that the mapping performance is mainly influenced by the connection pattern of the Sneptree instead of the mapping algorithm itself, the static model for the performance measurement is good enough to reveal the performance difference caused by different connection patterns.

Among the seven connection patterns, we found that the Exchange Sneptree (connection pattern (4)) is the best and the Self-Looping Sneptree is the worst. Apart from minor variants, the simulation results agree upon the following order of the mapping performance in terms of the load balancing: Exchange Sneptree, De Bruijn Network, Another Cyclic Sneptree, Linear Cyclic Sneptree, Cyclic Sneptree, Planar Cyclic Sneptree, and Self-Looping Sneptree.

From the simulation, we found that a Cyclic Sneptree doesn't always perform better than a noncyclic Sneptree in mapping an arbitrary unbalanced binary tree. A Cyclic Sneptree is ideal for mapping a linear array or a ring of arbitrary size. But the key property of a good connection pattern for our application is that no matter which node in the Sneptree is selected to be the root, we should end up with a satisfactory mapping for a complete binary tree. Among the seven connection patterns, we found that the Exchange Sneptree (connection pattern (4)) has the best performance in the mapping of a complete binary tree of arbitrary size with any node being the root. Nevertheless, we cannot assure that Exchange Sneptree is the best connection pattern for our application because there are still many connection patterns we have not investigated. Moreover, the Exchange Sneptree is not extensible; i.e., we can't build a bigger Exchange Sneptree recursively from smaller ones.

So far, the mapping performance refers only to the load- balancing factor. We shall also look at the other mapping criterion – the communication overhead. Our simulator calculates only the communication overhead for the tree links, not the horizontal links between sibling processes. Since the horizontal links will be the major source of the communication overhead due to the lack of corresponding links in the Sneptree, the simulation results may be too optimistic. Nevertheless, the figures give us a good guideline for comparing the performance of different mapping algorithms and different connection patterns.

From the simulation results, we found that the average communication overhead of the two sample programs is very low, ≤ 0.4. Both the mapping algorithms and the connection patterns affect the communication overhead (see Appendix B, Figure B.11 to Figure B.16). Mapping algorithm V1 usually has the least communication overhead because any cell and its single descendant are mapped onto the same node, which causes a "-1" communication overhead for such a father–son link. Mapping algorithms V3, V4 and V5 have no significant difference in terms of communication overhead. Regarding the connection patterns, the smaller the load balancing factor is, the higher the communication overhead. Therefore, the Exchange Sneptree and the Self-Looping Sneptree are the worst and the best, respectively, in terms of communication overhead.

Since the communication overhead is very low in our mappings and the difference between the best and the worst overhead is also very small, we decided that the load balancing factor is more important in determining the mapping performance. In Appendix ~~B, the complete simulation results of the load-balancing factor and the communication~~ overhead for the quicksort program is included. The result of the best mapping algorithm V5 for the program a(N) is also included for reference.

9.4 Mapping onto Other Networks

Is the Sneptree the best architecture for the Sync Model? In this section, we discuss the mapping of the process tree onto other interconnection networks, such as mesh, torus and Boolean n-cube. As we shall see, the Sneptree is superior to the other architectures for the mapping of the Sync Model.

The mapping performance from one graph to another usually depends on the similarity of the two graphs. The mapping performance is good when two graphs are similar. The process tree and the Sneptree are both unbounded, and the only difference between them is the fanout. Therefore, it is easy to transform one graph to the other graph. Mapping of the process tree onto other networks is more difficult. In the following, we take two different approaches: edge-to-edge mapping or one-to-one mapping.

9.4.1 Mesh and Torus

The process tree – an unbounded tree with arbitrary-fanout – can be mapped onto a fixed-size mesh or torus in two ways. The edge-to-edge mapping maps one edge of the process tree onto one edge of the mesh (Figure 9.11). The communication overhead is zero in this mapping, but the load of each node is very unbalanced. The nodes around the root are heavily loaded regardless of the size of the mesh. It is a common problem when mapping a logarithmic network onto a non-logarithmic network because in a non-logarithmic network, such as a mesh, the number of reachable nodes of a given node is linear to the distance from these nodes to the given node, while in a logarithmic network, such as the process tree, the number of reachable nodes is exponential to the distance.

The second approach, a one-to-one mapping, maps an H-structure tree onto the mesh (Figure 9.12). Because the H-tree is a two-dimensional layout for a binary tree, the process tree has to be transformed into a binary tree first, then mapped onto the mesh. The mapping of an H-tree onto a mesh has poor performance because: (1) The highest communication overhead is $\log_2 n - 1$, where n is the size of the H-tree. The longest path in the mapping is the one between the root and its two descendants. (2) The ratio of the unused node in the mesh to the size of the H-tree is very high, and it is higher when the H-tree is bigger. These unused nodes are either used as communication nodes or totally wasted. (3) This

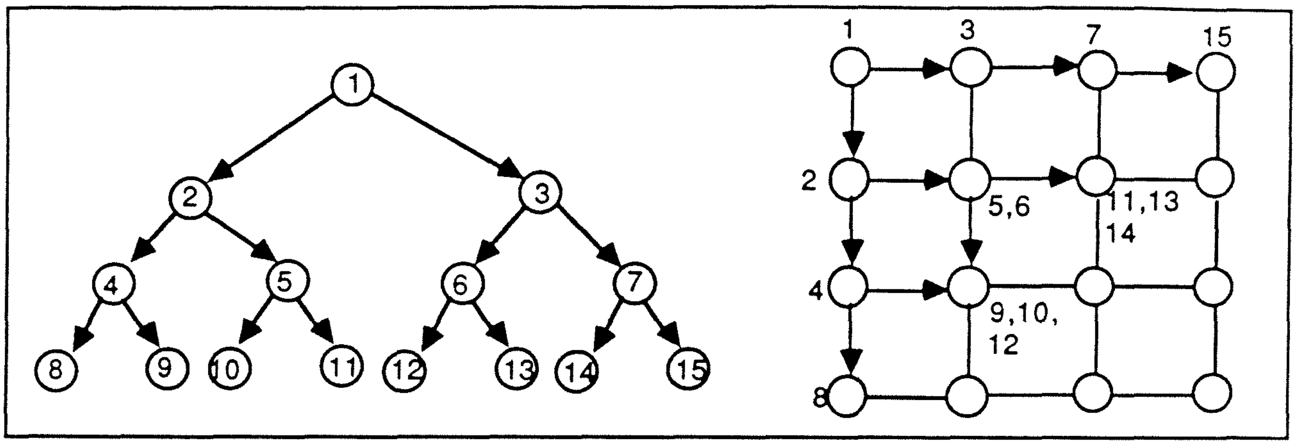

Figure 9.11. Edge-to-edge Mapping of a 4-level Binary Tree onto a 4×4 Mesh

mapping starts from mapping the leaf cells then up to the root. Since the process tree is dynamic, we have to map it starting from the root.

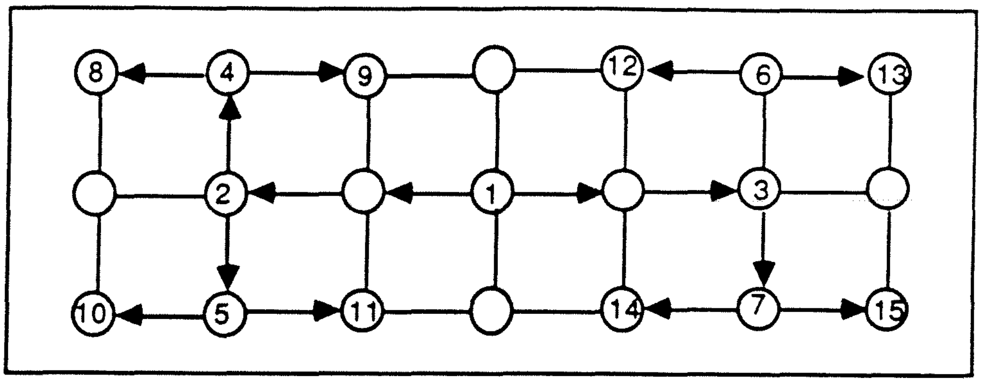

Figure 9.12. One-to-one Mapping of a 4-level Binary Tree onto a 7×3 Mesh

The mapping of the process tree onto the torus is essentially the same as mapping onto the mesh except that the boundary problems of the mesh never occur in the torus. We found that none of the mappings are satisfactory due to the diversity of the two graphs.

9.4.2 Boolean n-cube

The Boolean n-cube is another logarithmic network because the maximal distance of any two nodes in the cube is $\log_2 N$, where N is the size of the cube. Besides, the Boolean n-cube is a homogeneous network without a boundary. Therefore, it has similar properties to a Sneptree. Wu [56] suggests a method to map an n-level complete binary tree one-to-one onto an n-cube with the worst communication overhead being a constant (=1). Figure 9.13

shows the mapping from a three-level binary tree onto a 3-cube, where edge (1,2) in the tree is mapped onto a path of length 2 in the cube. This method can be modified to map a bigger binary tree onto a smaller cube and the result is load balanced. An arbitrary fanout tree has to be transformed into a binary tree first and then mapped onto the cube with Wu's method. Unfortunately, this mapping algorithm is also a bottom-up algorithm, which is not adequate in mapping the process tree, which dynamically grows and shrinks.

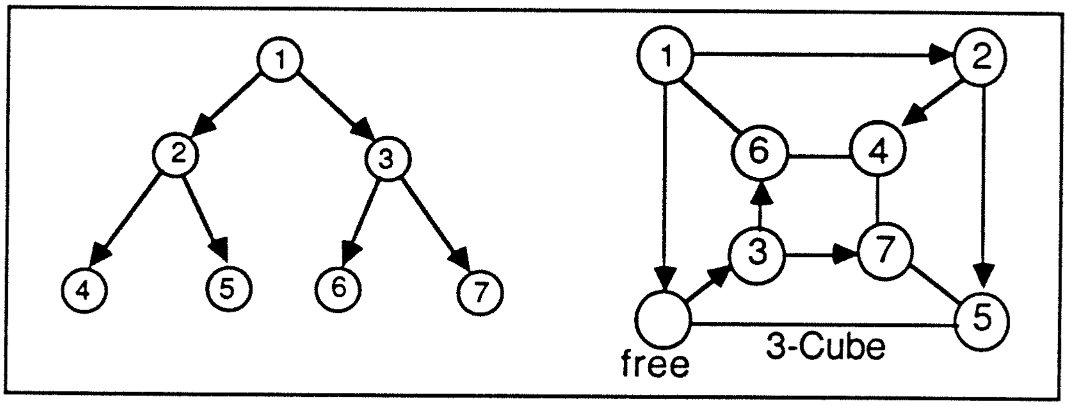

Figure 9.13. One-to-one Mapping from a 3-level Binary Tree onto a 3-cube

From the above discussion, we found that mapping of the process tree onto other networks also requires transforming the process tree into a binary tree first. Since the Sneptree is the best network for mapping a binary tree, the Sneptree is undoubtedly the best network for mapping a process tree.

9.5 Summary

In this chapter, we have presented a method to map the Sync Model onto a Sneptree. The mapping is in two steps: We first map the process tree which is an unbalanced tree, with arbitrary fanout and arbitrary size, onto an unbounded binary tree and then map such a binary tree onto a Sneptree. The first mapping step utilizes extra communication nodes to get a one-to-one mapping between two graphs. The second mapping is done automatically by following the connections of the Sneptree. Two mapping criteria have been defined for the measurement of the mapping performance: load balancing factor and communication overhead. A good mapping is a mapping with a low load-balancing factor and low communication overhead as well.

The mapping performance have been determined by simulation. Several similar mapping algorithms, with different load-balancing rules, are compared. Different connection patterns of the Sneptrees are also compared. We conclude that the mapping performance is influenced mainly by the connection pattern of the Sneptree instead of the mapping algorithm. Among different connection patterns, the Exchange Sneptree, in which all the outgoing links of the left half leaf nodes are directed to the nodes in the right subtree and vice versa, has the best load-balancing factor. We also found that a connection pattern which has a smaller load balancing factor usually has higher communication overhead. But the communication overhead is very low and the difference among different patterns is not significant. Therefore, we conclude that the Exchange Sneptree is the best connection pattern we found so far for the mapping of our Sync Model.

We also discussed the mapping from the process tree to a mesh, a torus or a Boolean n-cube. The mesh and the torus have poor performance in mapping a tree. The Boolean n-cube is a nice structure for tree embedding. A balanced mapping from a complete binary tree to an n-cube is possible. Nevertheless, the bottom-up mapping algorithm which results in a balanced mapping is not adequate for our application. From our analysis, the Sneptree is indeed an ideal architecture for the Sync Model.

Chapter 10

Conclusion

The goal of this thesis has been to explore an efficient parallel execution model for logic programming on a message-passing multiprocess system (e.g., ensemble machine), so that the inherent parallelism in a logic program, i.e., AND parallelism, OR parallelism and stream parallelism can be implemented efficiently.

Toward the realization of this goal, we designed a concurrent logic programming language, CLP and its computation model, the Sync Model. Like other extended logic programming languages, [9, 45, 55], CLP uses variable annotations to specify the producer and the consumer of a shared variable in the clause body and a commit operator to serialize the executions of two parts of the clause body. But the variable annotations and the commit operator in CLP are used optionally by the programmer to achieve a more efficient execution under the Sync Model. They are not required and do not change the semantics of the language. Therefore, although the Sync Model is designed for CLP, any Horn-clause program can be executed under the Sync Model.

The Sync Model is a multiple-solution data-driven model which generates all the solutions to an initial goal without explicit request. AND parallelism is implemented by constructing a dynamic data flow graph of the literals in the clause body with an ordering algorithm. OR parallelism is achieved by adding special synchronization signals to the stream of partial solutions and synchronizing the multiple streams with a merge algorithm.

The data flow graph constructed by the ordering algorithm results in a more efficient computation than other data flow models because of the following reasons: (1) Variable annotations added by the programmer help construct a more efficient graph than the graph constructed by simply assuming a left-to-right order. (2) Selective channels, which are used to filter out invalid inputs of a process that produces no outputs, help eliminate unnecessary computations in other sibling processes. (3) The goal process considered as a node in the graph receives the partial solutions of its output variables from its descendants and merges them to form the solutions to this goal. It is not necessary to send all the bindings to the goal. Thus, both the communication overhead and the work load of the goal process are reduced. And (4) the graph constructed dynamically reveals more parallelism than a static graph such as in [6].

The overhead of constructing the data flow graph is also less than other models such as [11], because the graph is constructed after each unification and modified by adding dynamic links when a variable is bound to a partially instantiated term. The overhead may be further reduced if in each OR process the data flow graph is constructed only for the first goal it receives and reused for the goals it receives later. Consider an OR process that receives several goals from its father AND process. The corresponding arguments in these goals are usually in the same modes; i.e., if an argument in one goal is a variable, then the corresponding arguments in the other goals are also variables. Therefore, the data flow graphs for different goals are usually the same. If we save the first data flow graph and the arguments of the first goal, we may reuse the same graph if the modes of the second goal's arguments match with the first goal. Hence, we need not perform the ordering algorithm after every unification. This improves the efficiency of the Sync Model significantly.

The merge algorithm that is used to synchronize the multiple input streams of a process is the first in the literature. It is superior to other methods for the implementation of OR parallelism in a message-passing multiprocessor system for the following reasons: (1) It replaces backtracking. (2) All the solutions will be returned without explicit request. In the backtracking models, such as [11], the first solution is generated through a "forward computation" and the rest of the solutions are generated through backtracking upon user's requests. (3) Only the partial solutions for the input variables of a process need to be sent to that process. We don't need to pass the entire binding environment around the sibling

AND processes and their father process, such as [28]. Thus, the communication overhead is greatly reduced. (4) This mechanism makes it possible to incorporate both AND parallelism and OR parallelism into one model without any form of backtracking.

The synchronization mechanism may be expensive when we execute a deterministic program. In a deterministic program, at most one solution is generated by a process and the synchronization is unnecessary. Therefore, in Chapter 6, we suggest distinguishing the Deterministic mode from the Nondeterministic mode. In the Deterministic mode, the merge algorithm and the Sync signals are eliminated so that no extra overhead caused by the synchronization is introduced.

In the Sync Model, we assume a finite AND/OR tree for a given program. If the execution of a program corresponds to an infinite AND/OR tree, or it generates infinite lists or infinite solutions, the Sync Model still works properly due to the following reasons: (1) Extended with stream parallelism and tail recursion, our model is able to handle infinite lists correctly. (2) With the base-case merge algorithm described in Section 5.2, we derive as many input combinations as possible even if one or more input streams are infinite. Therefore, within the storage limit, we will make maximum progress for a program.

The Sync Model is sound and complete. Soundness means it only generates correct solutions and completeness means it generates all the correct solutions. The soundness and completeness of the Sync Model are implied by the correctness of the merge algorithm, which is proved in Chapter 5.

A new class of interconnection networks, the Sneptree, is also presented in this thesis. The Sneptree is an example of logarithmic networks, which can simulate an unbounded complete binary tree optimally. Amongst different connection patterns of the Sneptree, some of them are regular and extensible so as to be well suited for VLSI implementation. The Sneptree is an ideal machine for the Sync model, in which a dynamic process tree is constructed. With a simple mapping algorithm, the Sync Model can be mapped onto the Sneptree with highly balanced load and low overhead.

Future Research

The Sync Model assumes that each process has enough memory to store the entire program. This is feasible when the program is small. But it is not practical when dealing with a program with a large database or knowledge base. One of the solutions to this problem is to partition the database into disjoint sets and distribute the divided sets over different processors [35,50]. The cost for such a technique is due to the extra communications in order to reach the processor which stores the clause needed in the inference process. Most AI applications are involved with a fairly big knowledge base; therefore, we have to solve this problem to make our Sync Model practically useful.

One suggestion is to have bigger granularity of each node and fewer nodes in a Sneptree. One copy of the program is stored in each node of the Sneptree, which is shared by all the processes mapped onto that node. If it is still too expensive to keep more than one copy of the program in the Sneptree, the program can be distributed over the Sneptree and later migrated to the node where the computation takes place. A process in the process tree may be used repeatedly to handle different data but the migration of the program happens only once for each process.

Another suggestion to handle a program with a big database is to search and unify clauses sequentially in one or a few OR processes instead of creating one OR process for each unifiable clause. Combinatorial explosion can be avoided in this way when solving a program with massive OR parallelism.

In this thesis, we proposed a parallel execution model for logic programming on the Sneptree. Many architecture issues are left open, such as the size of the Sneptree, the processor and the memory size of each node, the communication protocols, the interface to the outside world, etc.. In the meantime, much software support needs to be provided to make the machine possible, such as multiprocessing in a single processor, buffer and channel management, garbage collection, etc.. We would like to see a parallel machine, which executes logic programs effectively and efficiently, built based on the Sync Model.

Reference

[1] Bentley, J.L. and H.T. Kung, "A Tree Machine for Searching Problems", *Proceedings of the International Conference on Parallel Processing,*1979, pp. 257-266.

[2] Bic, L., "A Data-Driven Model for Parallel Interpretation of Logic Programs", *Proceedings of the International Conference on Fifth Generation Computer Systems 1984,*pp. 517-523, ICOT, 1984.

[3] Borgwardt, Peter, "Parallel Prolog using Stack Segments on Shared-Memory Multiprocessors", *1984 International Symposium on Logic Programming,*pp.2-11, Feb. 1984.

[4] Bowen, K.A., "Concurrent Execution of Logic", *First International Logic Programming Conference,*Sept. 82, pp. 26-30.

[5] Browning, S.A,. "The Tree Machine: A Highly Concurrent Computing Environment", Ph.D. thesis, Caltech, Computer Science, 1980.

[6] Chang, J.H., and D. DeGroot, "AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis", Dept. of Electrical Engineering and Computer Science, Univ. of Calif., Berkeley. Sept. 1984.

[7] Ciepielewski, Andrzej and Seif Haridi, "A Formal Model for OR-Parallel Execution of Logic Programs", *Information Processing 83,*edited by R.E.A. Mason, North-Holland, pp.299-305, IFIP 1983.

[8] Ciepielewski, Andrzej and Seif Haridi, "Control of Activities in the OR-Parallel Token Machine", *1984 International Symposium on Logic Programming,*pp.49-57, 1984.

[9] Clark, Keith L. and Steve Gregory, "A Relational Language for Parallel Programming", *Proceeding of 1981 Conference on Functional Programming Language and Computer Architecture,* pp.171-178

[10] Conery, John S. and D.F. Kibler, "Parallel Interpretation of Logic Programs", *Proceeding of the 1981 Conference on Functional Programming and Computer Architecture,*pp.163-170.

[11] Conery, John S., "The AND/OR Process Model for Parallel Interpretation of Logic Programs", Ph.D. Dissertation, TR204, University of California, Irvine, June 1983.

[12] Davis, Ruth E., "Logic Programming and Prolog: A Tutorial", *IEEE Software,*Vol. 2, No. 5, 1985, pp.53-62.

[13] DeGroot, Doug, "Parallel Execution of Logic Programming," Tutorial Notes, ACM Tutorial for Engineering Professionals, May 1985.

[14] DeGroot, Doug, "Alternate Graph Expressions for Restricted AND-Parallelism," *Compcom 85,*Spring, pp.206-210, Feb. 1985.

[15] Despain, A.M. and D.A. Patterson, "X-tree: A Tree Structured Multi-Processor Computer Architecture," *Conference Proceedings of the 5th Symposium on Computer Architecture,*1978, pp. 144-151.

[16] Eisinger N., S. Kasif, and J. Minker, "Logic Programming: A Parallel Approach," *First International Logic Programming Conference,*Sept. 82, pp. 71-77.

[17] Furukawa, K., K. Nitta, and Y. Matsumoto, "Prolog Interpreter Based on Concurrent Programming," *First International Logic Programming Conference,*Sept. 82, pp. 38-44.

[18] Goodman, J.R. and C.H. Sequin, "Hypertree: A Multiprocessor Interconnection Topology," Computer Science Technical Report #4227, Apr. 1981.

[19] Goto, Atsuhiro, Hidehiko Tanaka and Tohru Moto-oka, "Highly Parallel Inference Engine PIE - Goal Rewriting Model and Machine Architecture," *New Generation Computing 2,* pp.37-58, OHMSHA, LTD., 1984.

[20] Harary, F., *Graph Theory,* Addison-Wesley, Reading, Massachusetts, 1969.

[21] Hasegawa, R. and M. Amamiya, "Parallel Execution of Logic Programs based on Dataflow Concept," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984,*pp. 507-516, ICOT, 1984.

[22] Hong, J.W, K. Mehlhorn, and A.L. Rosenberg, "Cost Trade-offs in Graph Embeddings, with Applications," *JACM 30,*pp.709-728, 1983.

[23] Kowalski, R.A., "Predicate Logic as Programming Language," *Proc. IFIP 74,* North-Holland.

[24] Kowalski, R.A., *Logic for Problem Solving,* North Holland Inc., New York, 1979.

[25] Kung, H.T.,"Why Systolic Architectures?" *IEEE Computers 15(1),* January 1980, pp. 37-56.

[26] Leiserson, C.E., "Area-Efficient VLSI Computation," Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, Oct. 1981.

[27] Li, P., "The Tree Machine Operating System," TR:4618, Computer Science, Caltech, July 1981.

[28] Lindstrom, G., and P. Panangaden, "Stream-based Execution of Logic Programs," *1984 International Symposium on Logic Programming,* Feb. 1984.

[29] Lindstrom, G., "OR-Parallelism on Applicative Architectures," Lab. for Computer Science, Mass. Institute of Tech., Jan. 1984.

[30] Mago, Gyula A., "A Cellular Computer Architecture for Functional Programming," *COMPCON,* Spring 1980, pp. 179-187.

[31] Martin, A.J., "A Distributed Implementation Method For Parallel Programming," *Proceedings IFIP-Congress,* Oct. 1980.

[32] Martin, A.J., "The TORUS: an Exercise in Constructing a Processing Surface," *Proceedings of Second Caltech Conference on VLSI,* Jan. 1981, pp. 527-538.

[33] Martin, A.J. and J.L.A. van de Snepscheut, "Networks of Machines for Distributed Recursive Computations," TR:84:5147, Caltech, Computer Science, 1984.

[34] Mead, C. and M. Rem, "Cost and Performance of VLSI Computing Structures," *IEEE Journal of Solid State Circuits,* Vol. SC-14, No. 2, April 1979, pp. 455-462.

[35] Nakagawa, Hiroshi, "AND Parallel PROLOG with Divided Assertion Set," *1984 International Symposium on Logic Programming,* pp.22-28, Feb. 1984.

[36] Pereira, L.M. and L.F. Monteiro, "The Semantics of Parallelism and Co-routining in Logic Programming", in *Mathematical Logic in Computer Science 26,* Hungary, 1978, pp. 611-657.

[37] Pereira, L.M., and A. Porto, "Selective Backtracking for Logic Programs," Departamento de Informatica, CIUNL no. 1/80 University Nova de Lisboa

[38] Pereira, L.M., and R. Nasr, "DELTA-PROLOG: A Distributed Logic Programming Language," *Proceedings of the International Conference on Fifth Generation Computer Systems 1984,* pp. 283-291, ICOT, 1984.

[39] Robinson, J.A., *Logic Form and Function,* Edinburgh University Press, 1979.

[40] Rosenberg, A.L., "Data Encodings and Their Costs," *Acta Informatica 9,* pp. 273-292, 1978.

[41] Roussel, P., "PROLOG Manuel de Reference et d'Utilisation," Technical Report, University of Marseille, 1975.

[42] Schlumberger, M.L., "De Bruijn Communications Networks," Ph.D. Dissertation, Computer Science, Stanford University, 1974.

[43] Seitz, C.L., "Ensemble Architectures for VLSI – A Survey and Taxonomy," *Proceedings Conference on Advanced Research in VLSI,* pp. 130-135, 1982.

[44] Seitz, C.L., "The Cosmic Cube," *CACM,* 28(1), Jan. 1985, pp. 22-33.

[45] Shapiro, Ehud Y., "A Subset of Concurrent Prolog and Its Interpreter," TR-003, ICOT-Institute for New Generation Computer Technology, Jan, 1983, Japan.

[46] Shapiro, Ehud Y., "Systolic Programming: A Paradigm of Parallel Processing," Dept. of Applied Math. The Weizmann Institute of Science, CS84-21, August 1984.

[47] Shaw, D.E., "The NON-VON Supercomputer," Dept. of Computer Science, Columbia Univ. Aug. 1982.

[48] Shaw, D.E., "NON-VON's Applicability to Three AI Task Areas," *Proceedings of the Ninty International Joint Conference on Artificial Intelligence,* 1985, pp.61–72.

[49] van de Snepscheut, J.L.A., "Mapping a Dynamic Tree on a Fixed Graph," unpublished article, Feb. 1981.

[50] Stolfo, S.J., and D.E. Shaw, "DADO: A Tree-Structured Machine Architecture for Production Systems," *Proceedings of the National Conference on Artifiial Intelligence,* 1982, pp.242–246.

[51] Umeyama, Shinji and Koichiro Tamura, "A Parallel Execution Model of Logic Programs," *the 11-th Annual International Symposium on Computer Architecture,* pp.349-355, June 1983.

[52] Warren, David S., "Implementing Prolog - Compiling Predicate Logic Programs," D.A.I. Research Reports 39,40, University of Edinburgh, 1977.

[53] Warren, David S., Mustaque Ahamad, Saumya K. Debray and L.V. Kale, "Executing Distributed Prolog Programs on a Broadcast Network," *1984 International Symposium on Logic Programming,* pp.12-21, 1984.

[54] Wise, Michael J., "A Parallel PROLOG: the construction of a data driven model," *Proceedings of the Symposium on LISP and Functional Programming,* pp. 56-66, ACM, August, 1982.

[55] Wise, Michael J., "EPILOG: re-interpreting and extending Prolog for a multiprocessor environment," in *Implementation of PROLOG,* edited by J.A. Campbell, pp.341-351, Ellis Horwood, 1984.

[56] Wu, A.Y., "Tree Network Embedding Into Hypercubes," *Journal of Parallel and Distributed Computing 2,* pp.238-249, 1985.

[57] Yasuhara, Hiroshi, and Kazuhiko Nitadori, "ORBIT: A Parallel Computing Model of Prolog," *New Generation Computing 2,* pp.277-288, OHMSHA, LTD. 1984.

Appendix A

Simulation Results for the Ordering Algorithm

The ordering algorithm is written in DEC-20 Prolog, which is used to measure the performance of the algorithm. Seven examples are selected and each of them has a different data flow graph. The first two examples are the quicksort and the query with students database described in Section 4.3. The third example is a map coloring program with four nodes in the map. The fourth and fifth examples are programs which have two multiple paths in their data flow graph either side by side or top-down. The last two examples are the two worst cases for the first step of the ordering algorithm: one has eight independent nodes and the other has eight nodes serially connected into a chain.

In the simulation result, "number of procedure calls:fire" is the number of table accesses in order to move all the literals from UPL to FPL. It is the figure for the performance of the first step of the ordering algorithm: the construction of the data flow graph. The following "number of procedure call:check" shows the number of stack accesses in order to detect a multiple path for a given node. If the value is 1, the given node has only one descendant so that no stack is built up for this node. If the value is more than 1, "Length of the stack" before that line shows the size of the stack upon each stack access. The sum of the procedure calls for "check" gives the figures in the column of "searching" in Table 4.15. The result process list (FPL) is shown in X. Each element in X consists of the literal with the variable annotated, the process id and a flag for Sync generator – true for a Sync generator, false otherwise. The data flow graph including the father process (with id "0") is followed ~~after each example except Examples 1 and 2, which can be found in Figure 4.6 and Figure~~ 4.13.

The program is also included following the simulation results.

```
Prolog-10  version 3

| ?- ['order.pl'].

order.pl consulted   3280 words      1.52 sec.

yes
% ******* Example 1: quicksort ************
| ?- test1(X).
Number of procedure call:fire is 4
Length of the stack: 3
Length of the stack: 4
number of procedure call:check is 3

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1


X = [[split(2,[1,3],Smaller!,Larger!),1,true],
    [sort(Smaller?,Sorted1!),2,false],
    [sort(Larger?,Sorted2!),3,false],
    [append(Sorted1?,[2|Sorted2?],Sorted!),4,false]]

yes
```

```
% ******* Example 2: database query for students ************
| ?- test2(X).
Number of procedure call:fire is 14
Length of the stack: 4
Length of the stack: 7
number of procedure call:check is 3

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

Length of the stack: 4
Length of the stack: 5
number of procedure call:check is 3

number of procedure call:check is 1

number of procedure call:check is 1


X = [[student(S!,C1!),1,true],
    [course(C1?,D1!,R!),2,false],
    [professor(P!,C1?),3,false],
    [student(S?,C2!),4,false],
    [C1? =\=C2?,5,true],
    [course(C2?,D2!,R?),6,false],
    [professor(P?,C2?),7,false]]

yes
```

```
% ******* Example 3: map coloring with four nodes ************
| ?- test3(X).
Number of procedure call:fire is 10
Length of the stack: 5
number of procedure call:check is 2

number of procedure call:check is 1

number of procedure call:check is 1

Length of the stack: 3
number of procedure call:check is 2

Length of the stack: 3
Length of the stack: 3
Length of the stack: 3
number of procedure call:check is 4


X = [[next(A!,B!),1,true],
    [next(A?,C!),2,false],
    [next(A?,D!),3,false],
    [next(B?,C?),4,true],
    [next(B?,D?),5,false]]

yes
```

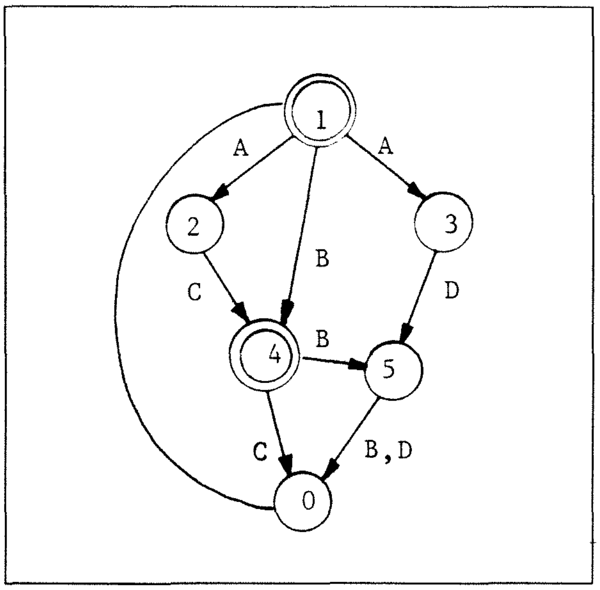

Figure A.1. The Data Flow Graph of Example 3

```
% ******* Example 4: two side by side multiple paths ************
| ?- test4(X).
Number of procedure call:fire is 7
Length of the stack: 3
Length of the stack: 4
number of procedure call:check is 3

number of procedure call:check is 1

Length of the stack: 3
Length of the stack: 4
Length of the stack: 5
Length of the stack: 6
number of procedure call:check is 5

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1


X = [[a(x,X!),1,true],
    [c(X?,X1!),2,false],
    [b(x,Y1!,Y2!),3,true],
    [d(X?,Y1?,Z!),4,false],
    [f(X1?,Z?),5,false],
    [e(Y2?,Y!),6,false],
    [g(Z?,Y?,b,T!),7,false]]

yes
```

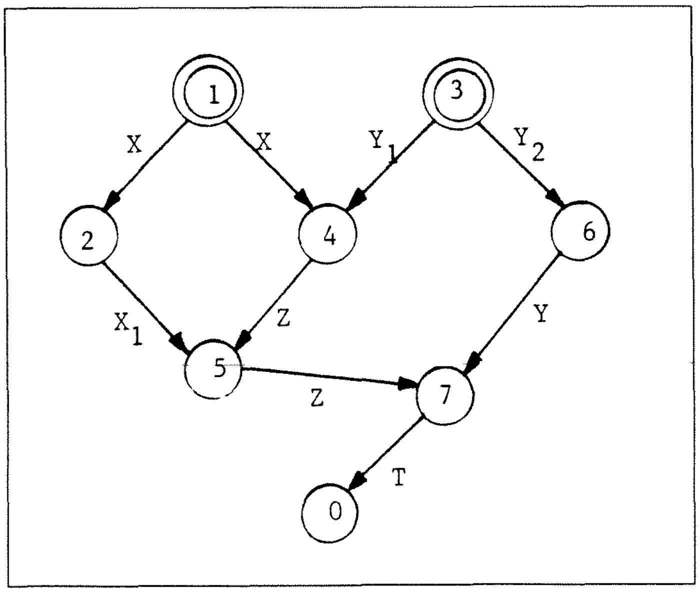

Figure A.2. The data flow graph of Example 4

```
% ******* Example 5: two top and bottom multiple paths ************
| ?- test5(X).
Number of procedure call:fire is 15
Length of the stack: 3
Length of the stack: 4
number of procedure call:check is 3

number of procedure call:check is 1

number of procedure call:check is 1

Length of the stack: 3
Length of the stack: 4
number of procedure call:check is 3

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1


X = [[a(a,X!),1,true],
     [b(X?,Y!),6,false],
     [c(X?,Z!),7,false],
     [d(Y?,Z?,U1!,U2!),5,true],
     [e(U1?,V!),3,false],
     [f(U2?,W!),4,false],
     [g(V?,W?,S!),2,false]]

yes
```

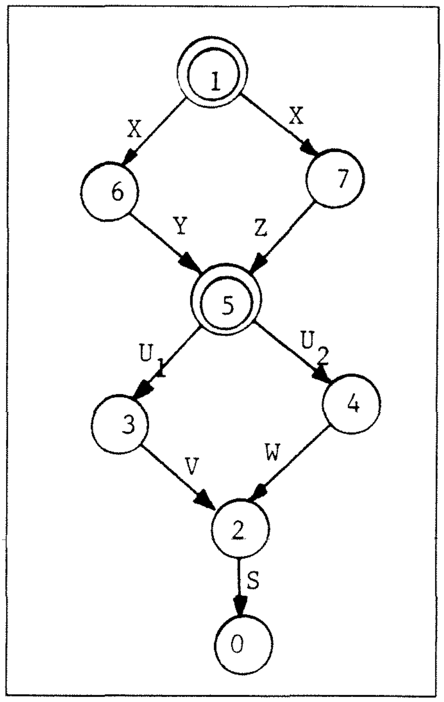

Figure A.3. The data flow graph of Example 5

```
% ******* Example 6: 8 independent nodes ************
| ?- test6(X).
Number of procedure call:fire is 44
number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1


X = [[a(X1!),1,false],
    [b(X2!),2,false],
    [c(X3!),3,false],
    [d(X4!),4,false],
    [e(X5!),5,false],
    [f(X6!),6,false],
    [g(X7!),7,false],
    [h(X8!),8,false]]

yes
```

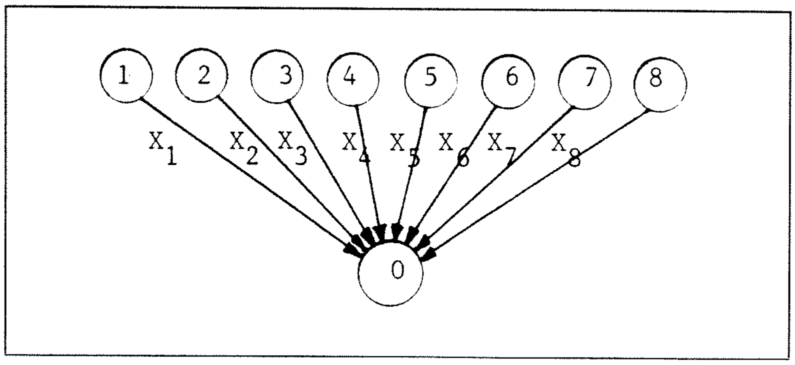

Figure A.4. The Data Flow Graph of Example 6

```
% ******* Example 7: 8 nodes in a chain ************
| ?- test7(X).
Number of procedure call:fire is 36
number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1

number of procedure call:check is 1


X = [[h(x,X2!),8,false],
     [g(X2?,X3!),7,false],
     [f(X3?,X4!),6,false],
     [e(X4?,X5!),5,false],
     [d(X5?,X6!),4,false],
     [c(X6?,X7!),3,false],
     [b(X7?,X8!),2,false],
     [a(X8?),1,false]]

yes

| ?- halt.
```

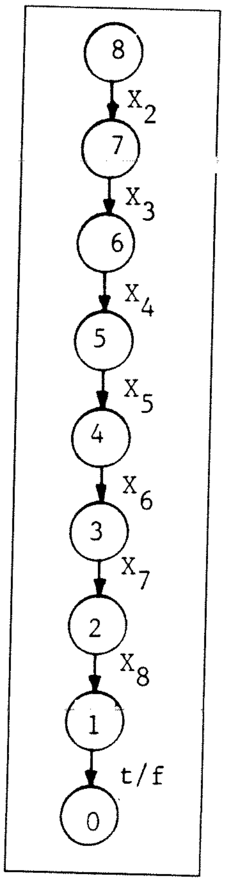

Figure A.5. The data flow graph of Example 7

```
% ******** The Ordering Algorithm ****************
:-op(200,xf,[?,!]).  % input/output annotations

% main program: GoalVar is a list of uninstantiated variables in the goal
%       UPL is a list of AND literals
%       FPL is a list of fired AND literals, their ids and flags for Sync Generators
ordering(GoalVar,UPL,FPL) :- init(GoalVar),
                    order(UPL,UPL1,1),
                    fire(UPL1,[],FPL1,0,X,empty),
                    refine(FPL1),
          markSync(FPL,FPL).

% put variables in the goal into the channel table.
% The channel table is a global data structure stored in the form of
% ct(Var,Producer,Consumers).
init([]).
init([X|L]):-assert(ct(X,[],[0])), init(L).

% step (1.a): first scan of UPL and initiation of the channel table.
% CT is stored as a predicate ct(Var,Producer,Consumers).
order([],[],_).
order([G|L],[[G,N]|L1],N):-G=..[_|Args],
                   checkArg(Args,N), N1 is N+1,
                   order(L,L1,N1).


checkArg([],_).
checkArg([X|L],N):-invar(X,X1),
                   (retract(ct(X1,Y,Z));Y=[],Z=[]),
                   assert(ct(X1,Y,[N|Z])),!,
                   checkArg(L,N).

checkArg([X!|L],N):-(retract(ct(X,Y,Z));Z=[]),
                   assert(ct(X,N,Z)),!,
                   checkArg(L,N).

checkArg([X|L],N):- + (atomic(X)),
                   X=..[F|Args],
                   checkArg(Args,N),
                   checkArg(L,N).

checkArg([_|L],N):-checkArg(L,N).

%Step (1.b) to (1.d): fire one literal and add annotations to its arguments.
fire([[G,N]|L1],L2,[[G1,N]|L3],C,X,T):- G=..[F|Args],
                   C1 is C+1,                                  %Count
                   checkVar(Args),!,
                   addAnnot(Args,NewArgs,N),
                   G1=..[F|NewArgs],
                  (T=nonempty ->
                   fire(L1,L2,L3,C1,X,true);
```

```
                fire(L1,L2,L3,C1,X,T)).

fire([G|L1],L2,L3,C,X,T):-C1 is C+1,                          %Count
                 append(L2,[G],L4),fire(L1,L4,L3,C1,X,nonempty).

fire([],L2,L3,C,X,true):- fire(L2,[],L3,C,X,empty).

fire([],[[G,N]|L2],[[G1,N]|L3],C,X,nonempty):- G=..[F|Args],
                 C1 is C+1,                                    %Count
                 addAnnot(Args,NewArgs,N),
                 G1=..[F|NewArgs],
                 fire(L2,[],L3,C1,X,empty).

fire([],[],[],C,C,_):-write('Number of procedure call:fire is '), write(C),
                       nl.

checkVar([X|_]):-  + ( X=Y?; X=Y!; variab(X)).
checkVar([X|_]):-invar(X,Y),ct(Y,Z,_),  +(Z=[]).
checkVar([_|L]):-checkVar(L).

% add ! to all the variables without annotation in L1 and update CT.
addAnnot([],[],_).
addAnnot([X|L],[X|L1],N):- ( X=Y? ; X=Y!),addAnnot(L,L1,N).
addAnnot([X|L],[Y|L1],N):- + (atomic(X)), X=..[F|Args],
                       addAnnot(Args,NArgs,N), Y=..[F|NArgs],
                       addAnnot(L,L1,N).
addAnnot([X|L],[X|L1],N):- + (variab(X)), addAnnot(L,L1,N).
addAnnot([X|L],[X!|L1],N):-retract(ct(X,[],Z)),
                       delete(N,Z,Z1),
                       assert(ct(X,N,Z1)),
                       addAnnot(L,L1,N).

addAnnot([X|L],[X?|L1],N):-addAnnot(L,L1,N).

delete(X,[X|L],L).
delete(X,[Y|L],[Y|L1]):-delete(X,L,L1).

% Step (2): refine the data flow graph by adding output variables to the processes
% that have no output links.
refine([]).
refine([[G,N]|L]):-G=..[_|Args],
                 (input(Args,In) -> make_channel(In,N,false,T);
                                    T=true),
                 (T=false -> assert(ct(tf,N,0));true), % add true/false channel
                 refine(L).

% True if there are no output variables in the first argument, or the output variable
% has no consumers at all.  If true, the second argument contains all the input variables.
input([],[]).
input([X?|L],[X|L1]):-!,input(L,L1).
input([X!|L],L1):-!,ct(X,_,[]),!,input(L,L1).
input([X|L],L1) :- + (atomic(X)),
                X=..[F|Args],
                !,input(Args,L2),
                !,input(L,L3),
                append(L2,L3,L1).
input([X|L],L1):-!,input(L,L1).

% Create the selective channels for a set of variables in the first argument.
% If no selective channels are created, the last argument is set to false.
make_channel([],_,T,T).
make_channel([X|L],N,T,T1):-ct(X,P,C),
                        delete(N,C,C1),
                        (noempty(C1) -> retract(ct(X,P,C)),
                                        assert(ct(X,P,[N])),
                                        assert(ct(X,N,C1)),
                                        T2=true; T2=T),
                        make_channel(L,N,T2,T1).

noempty([_|_]).

% Step (3): detect multiple paths and mark the starting process of a multiple path
% as a SYNC generator
% the stack is a global data in the form of st(List).
markSync([],[]).
markSync([[G,N]|L],[[G,N,Found]|L1]):- assert(st([N])), %create stack
                             check(1,Found,X),
                             write('number of procedure call:check is '),
                             write(X), nl,nl,
                             markSync(L,L1).

% Check if the descendants of the K-th element of the stack are already in the stack.
% X is the count of procedure calls for check.  Found is true if a multiple path is found.
check(K,Found,X):-retract(st(L)),
                elmt(L,K,N),
                assert(bag([])), % bag(List) temporarily stores the descendants
                (ct(_,N,Y),
                 retract(bag(B)),
                 asserta(bag([Y|B])), fail;
                 retract(bag(B))),
                flatten(B,B1),
                length(B1,L1),
                ((K=1,(L1=1;L1=0)) -> Found=false,X=K;
                        (intersect(B1,L) -> Found=true,X=K;
                        append(L,B1,NL),
                        assert(st(NL)),
                        K1 is K+1,
                        length(NL,Lg),
                        write('Length of the stack: '), write(Lg), nl,
                        (K1 =< Lg ->
                          check(K1,Found,X);
                          retract(st(_)),Found=false, X=K1
                        ))
                ).
```

```
flatten([],[]).
flatten([[]|L1],L2):-flatten(L1,L2).
flatten([[X|L]|L1],[X|L2]):-flatten([L|L1],L2).

intersect([X|L1],L2):-member(X,L2),!.
intersect([_|L1],L2):-intersect(L1,L2).

member(X,[X|_]):-!.
member(X,[_|L]):-member(X,L).

append([],L,L).
append([end],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).

%elmt(L,N,R): R is the Nth element of list L.
elmt([X|_],1,X):-!.
elmt([_|L],N,X):-integer(N) -> N1 is N-1, elmt(L,N1,X);
                            elmt(L,N1,X), N is N1+1.

variab(X):-atom(X), name(X,[C|_]),!, C=<90, C>=65.

invar(X?,X):-!.
invar(X,X):-variab(X).

retractall(F):-retract(F),fail;true.

% Inputs for the seven examples
test1(X):- G=['Sorted'],
        L=[     split(2,[1,3],'Smaller','Larger'),
        sort('Smaller'?,'Sorted1'),
        sort('Larger'?,'Sorted2'),
        append('Sorted1'?,[2|'Sorted2'],'Sorted')],
        ordering(G,L,X),retractall(ct(_,_,_)).

test2(X):- G=['S','P'],
        L=[  student('S','C1'),
                course('C1','D1','R'),
                professor('P','C1'),
                student('S','C2'), 'C1'= ='C2',
                course('C2','D2','R'),
                professor('P','C2')],
          ordering(G,L,X),retractall(ct(_,_,_)).

test3(X):- G=['A','B','C','D'],
        L=[next('A','B'),
          next('A','C'),next('A','D'),next('B','C'),next('B','D')],
          ordering(G,L,X),retractall(ct(_,_,_)).


test4(X):- G=['T'],
        L=[  a(x,'X'),c('X','X1'),b(x,'Y1','Y2'),d('X','Y1','Z'),
                  f('X1','Z'),e('Y2','Y'),g('Z','Y',b,'T')],
              ordering(G,L,X),retractall(ct(_,_,_)).

test5(X):- G=['S'],
            L=[  a(a,'X'),g('V','W','S'),e('U1','V'),f('U2','W'),
                  d('Y','Z','U1','U2'),b('X','Y'),c('X','Z')],
              ordering(G,L,X),retractall(ct(_,_,_)).

test6(X):- G=['X1','X2','X3','X4','X5','X6','X7','X8'],
            L=[ a('X1'),b('X2'),c('X3'),d('X4'),e('X5'),f('X6'),g('X7'),
                 h('X8')],
              ordering(G,L,X),retractall(ct(_,_,_)).

test7(X):- G=['X8'],
            L=[a('X8'),b('X7','X8'),c('X6','X7'),d('X5','X6'),e('X4','X5'),
                  f('X3','X4'),g('X2','X3'),h(x,'X2')],
              ordering(G,L,X),retractall(ct(_,_,_)).
```

Appendix B

Simulation Results for the Mapping Performance

In this appendix, two sample programs, *quicksort* and *a(N)*, that are used to test the mapping performance are described. The arguments of the programs shown in Figure B.1 are not the real arguments of the original program. The arguments are used to represent the number of recursive calls, in other words, the index of iterative loops. Observe that in the quicksort program in Figure 4.3, *sort* is recursively defined in the first argument $[X|Unsorted]$; therefore, the argument of $sort(N)$ in Figure B.1 refers to the length of $[X|Unsorted]$. Literal *sort* calls four literals: *split* is recursively defined in its second argument $[Unsorted]$, which is of length $(N-1)$; two *sort* are then called to sort two shorter lists and the total length of the two lists is $(N-1)$. We introduce a new variable N1 to represent the length of `Smaller` in the first `sort`, where N1≤N. N1 can be given an arbitrary number by the user to simulate data dependence. Finally, $append(N1-1)$ is called, where $(N1-1)$ is the length of the first argument in the definition of append because it is defined recursively in the first argument. The data dependence property cannot be represented adequately in this transformation. In the original program, there are three clauses for `split`; one of the first two will be chosen based on the values of the arguments. In the following program, it is difficult to represent such a data-dependent choice and therefore, we always select the first clause of $split(N)$. Such a program with no data dependence provides a very close approximation to the real process tree.

The second sample program $a(N)$ is selected to demonstrate both AND and OR parallelism. The definitions of a, b and c are examples of OR parallelism. For instance, the two clauses for $a(N)$ can be executed in parallel. Besides, predicates b, c, and d are recursively defined.

The transformation of the data-independent program into the input form of the sim-

```
% quicksort
sort(0).
sort(N):-split(N-1),sort(N1-1),sort(N-N1),append(N1-1). % N1≤N
split(0).
split(N):-split(N-1).
split(N).
append(0).
append(N):-append(N-1).

% a(N)
a(N):-b(N-1),c(3).
a(N):-d(N-2).
b(N):-e(0),b(N-1).
b(N):-d(N-1).
b(0).
c(0).
c(N):-c(N-1).
c(N).
d(0).
d(N):-e(0),f(0),d(N-1).
e(0).
e(0).
f(0).
```

Figure B.1. The Data-Independent Programs: sort(N) and a(N)

ulation program is straightforward. Figure B.2 shows the program input of the simulator. Each clause in the program is defined by a unit clause with predicate `program`. The first argument of `program` is the clause head and the second argument is a list of all the OR branches.

Based on the definition above, the process tree of $sort(4)$ constructed by the simulator is shown in Figure B.3. The process tree of $a(4)$ is also shown in Figure B.4. The two trees have 74 and 93 nodes, respectively. They are both unbalanced and their fanouts vary from 1 to 4. An AND node is marked by the goal it contains and an OR node is marked by "OR." The communication channels between sibling AND nodes are not simulated here because we are mainly interested in the load balancing of the mapping. Besides, the channels between sibling AND processes are simulated by the route from one AND process directed to their lowest common ancestor, then to the other AND process. Therefore, the communication overhead for the horizontal channels is consistent with that for the tree links. The simulator computes the average communication overhead for the tree links, which is good enough for comparison of different algorithms or different connections.

```
% quicksort
program(sort(0),[or,or]).
program(sort(N),[or,[or,split(N-1),sort(N1-1),sort(N-N1),append(N1-1)]]).
program(split(0),[or,or,or]).
program(split(N),[or,[or,split(N-1)],or]).
program(append(0),[or,or]).
program(append(N),[or,[or,append(N-1)]]).

% an example with AND and OR parallelism
program(a(N),[[or,b(N-1),c(3)],[or,d(N-2)]]).
program(b(0),[or,or,or]).
program(b(N),[[or,e(0),b(N-1)],or,[or,d(N-1)]]).
program(c(0),[or,or,or]).
program(c(N),[or,[or,c(N-1)],or]).
program(d(0),[or,or]).
program(d(N),[[or,e(0),f(0),d(N-1)],or]).
program(e(0),[or]).
program(f(0),[or,or]).
```

Figure B.2. The Program Input of the Simulator

Figures B.5 to B.8 show the load-balancing factors of the quicksort program in different connection patterns. Three sizes of the Sneptree, 15 nodes, 31 nodes, and 63 nodes, seven different connection patterns, (1) to (7) (see Section 9.3), and five different mapping algorithms, (V1) to (V5) (also see Section 9.3) are compared. Each load-balancing factor in the figures is the average of the LF's of four process trees for quicksort program, i.e., $sort(4), sort(5), sort(6)$, and $sort(7)$. The total number of processes in these four process trees are 74, 104, 132, and 153, respectively. Notice that the algorithms (V2) and (V3) result in the same mapping in this quicksort program because any AND node in the process tree has at least two descendants. Figure B.9 shows the load-balancing factors of $a(N)$ in different connection patterns with mapping algorithm V5, where V5 is the best mapping algorithm for $a(N)$ according to the simulation results. Figure B.10 shows the load-balancing factors of the quicksort program for four different mapping algorithms. Each LF in this figure is the average of LF's on three sizes of the Sneptree. Similarly, Figures B.11 to B.16 show the average communication overhead of the program $sort(N)$ and $a(N)$ in different connection patterns.

For the quicksort program, since there is no AND process with a single OR descendant, algorithms (V2) and (V3) result in the same mapping. Roughly speaking, algorithm (V5) performs the best, (V1) the worst, and (V2), (V3), and (V4) equally well for the quicksort

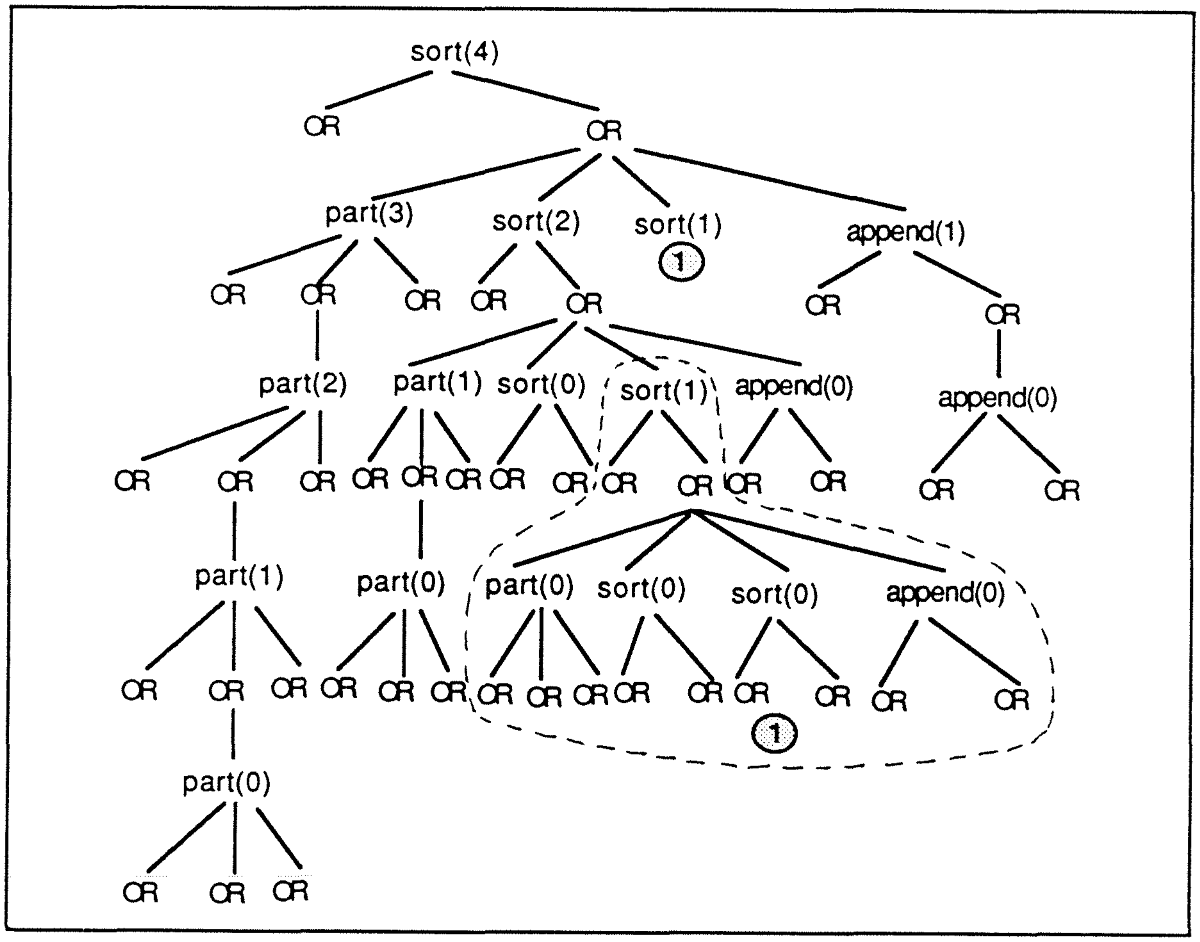

Figure B.3. The Process Tree for *sort*(4)

program with 4,5,6 or 7 elements to be sorted. For the second sample program, $a(N)$, the performance of the five mapping algorithms can be ordered as (V3), (V5), (V2), (V1), and (V4), where (V3) performs the best and (V4) the worst. It is interesting to see that a mapping algorithm with load- balancing heuristics doesn't perform better than the one without load balancing, such as (V4) (with load balancing) and (V3) (without load balancing) Another observation is that the two algorithms with load balancing ((V4) and (V5)) are significantly different; i.e., (V5) usually performs better than (V4). These phenomena can be explained, because when mapping the N descendants, $N > 2$, of a given cell in the process tree onto the subtree of a given node in the Sneptree, the two direct descendants of the given node are likely to be used as communication nodes, and the load factors do not increase in this mapping. Therefore, the load-balancing heuristics based on the load factors

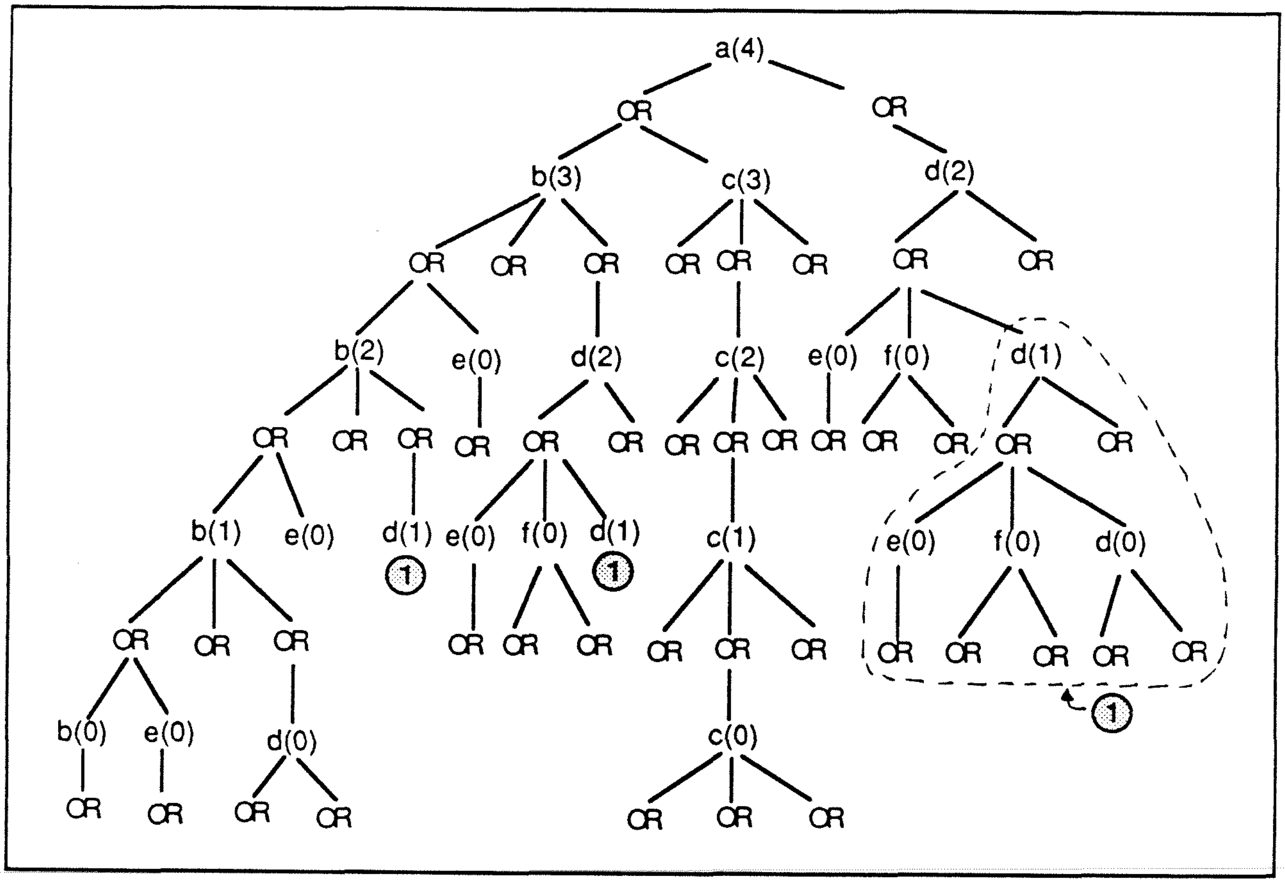

Figure B.4. The Process Tree for $a(4)$

of the two direct descendants doesn't help much in load balancing. Only when N=3, we try to map two of the cells to one descendant of the given node and the other cell to the other descendant. With algorithm (V5), we increase the load factor of the lighter loaded descendant by one, but we don't increase the load factor of the other descendant. This indeed helps to balance the load of the two descendants; therefore, (V5) is always better than (V4).

The simulation results don't favor algorithm (V1), in which we map a process and its ~~single descendant to the same node. From a practical viewpoint, such a mapping is more~~ efficient than others because having a process with a single descendant implies that there is no parallelism whatsoever, and mapping them into two different nodes simply introduces communication overhead but no extra speed. The mapping performance we measured is based on the entire process tree. We ignored the fact that not all processes in the process tree are active at the same time. A more precise performance evaluator should take the time factor into account, which involves a more complex analysis of the program and the construction of the process tree.

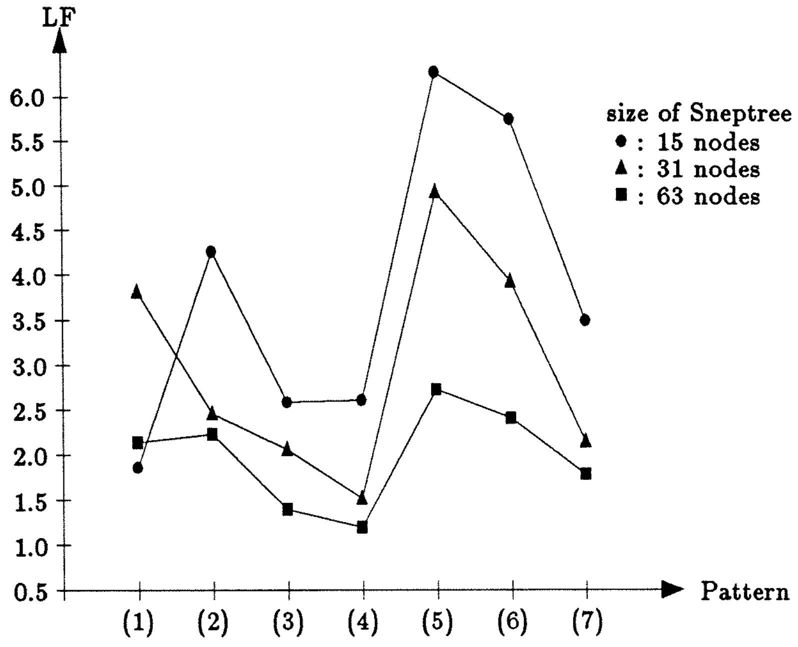

Figure B.5. The Load-balancing Factor LF of *sort*(N) with Algorithm V1

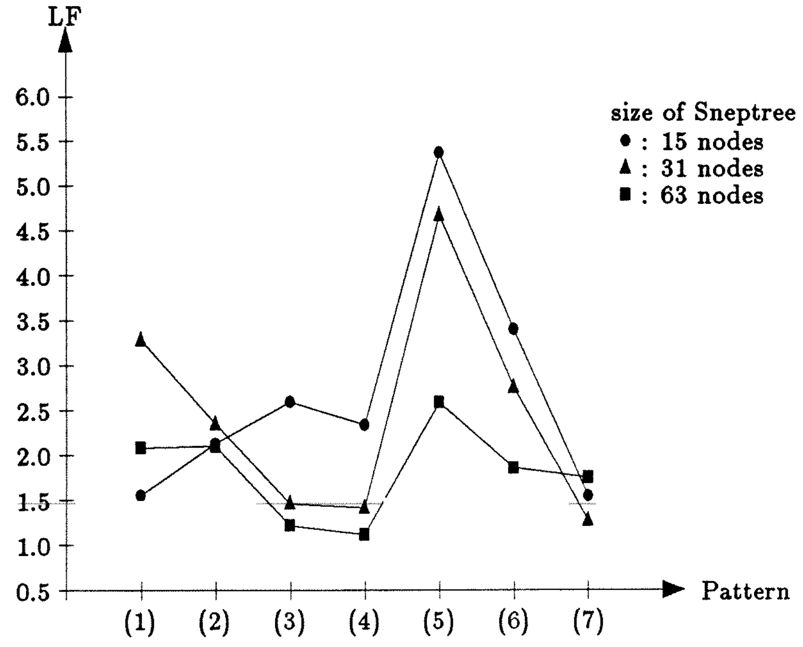

Figure B.6. The Load-balancing Factor LF of *sort*(N) with Algorithm V2

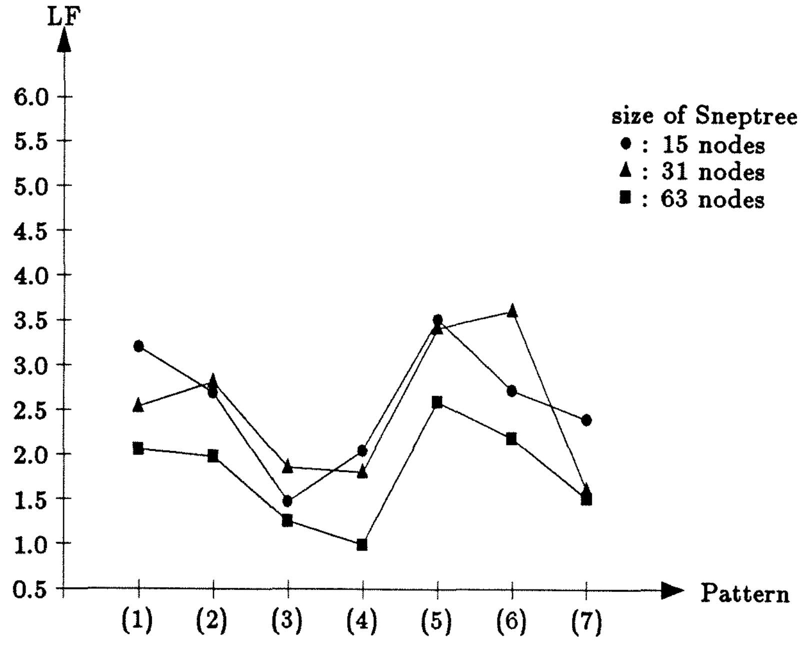

Figure B.7. The Load-balancing Factor LF of *sort*(N) with Algorithm V4

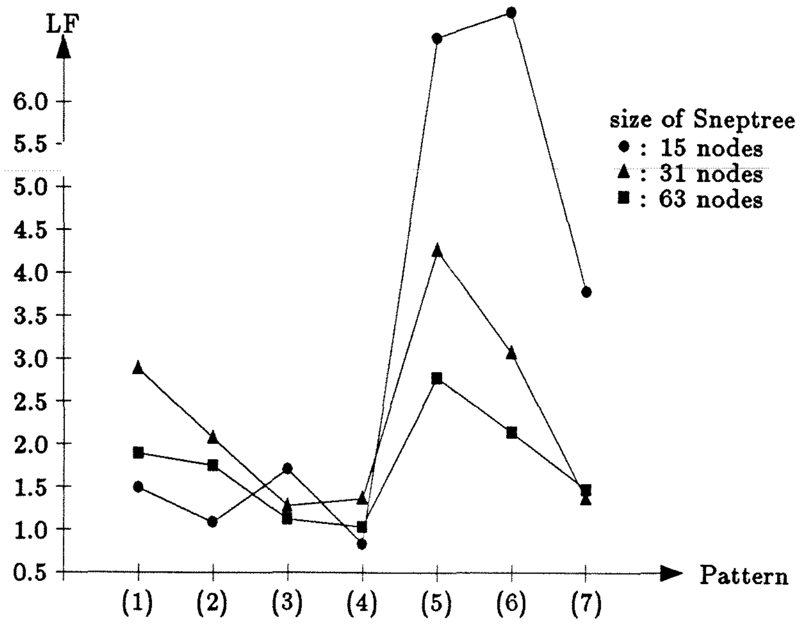

Figure B.8. The Load-balancing Factor LF of *sort*(N) with Algorithm V5

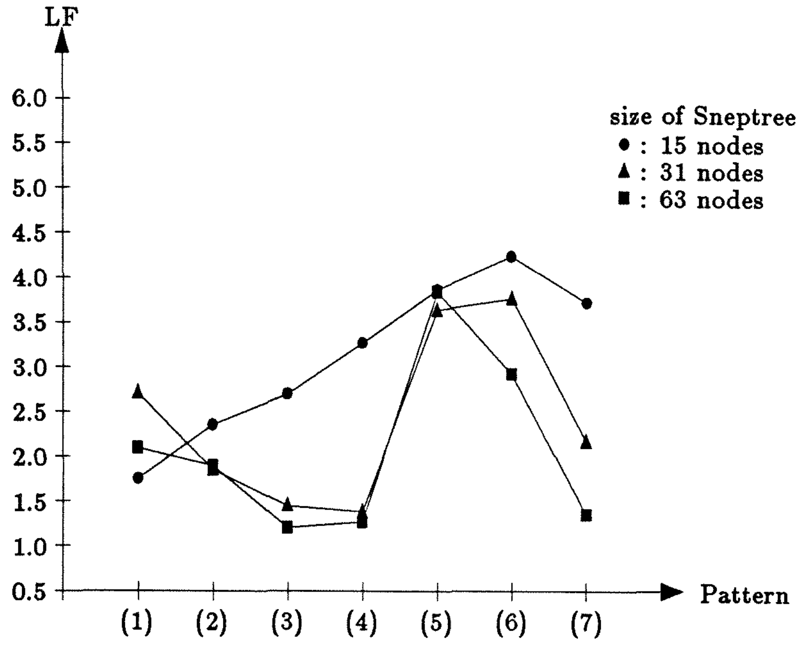

Figure B.9. The Load-balancing Factor LF of $a(N)$ with Algorithm V5

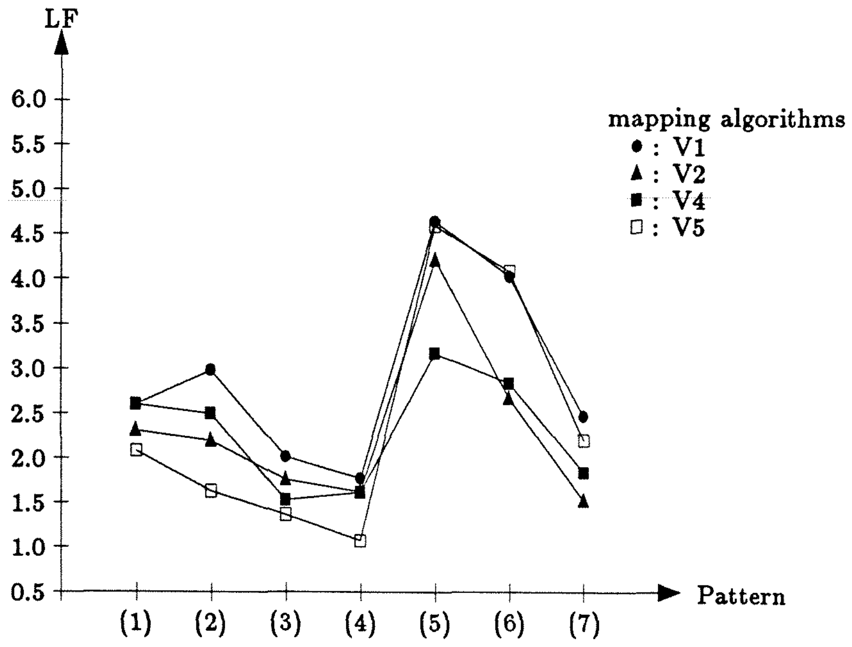

Figure B.10. The Load-balancing Factor LF of *sort*(N) for Different Algorithms

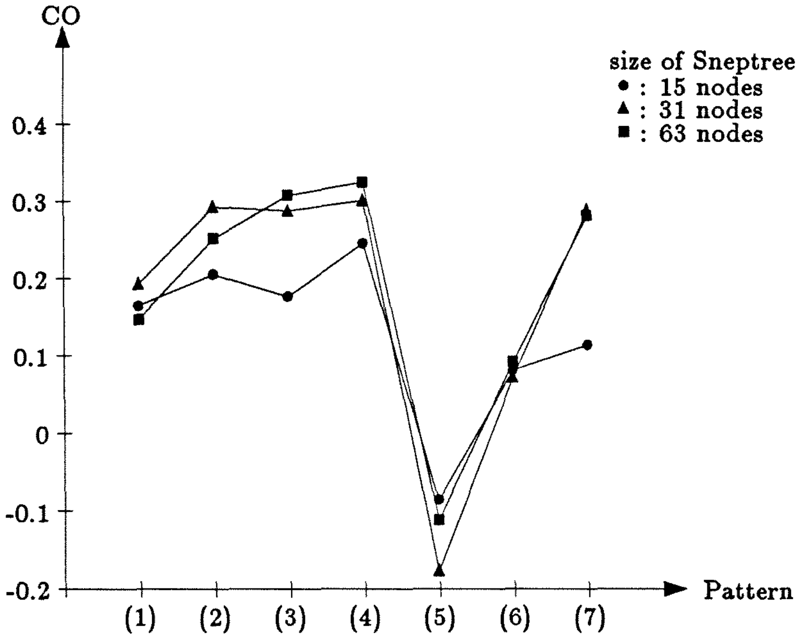

Figure B.11. The Average Communication Overhead CO of *sort*(N) with Algorithm V1

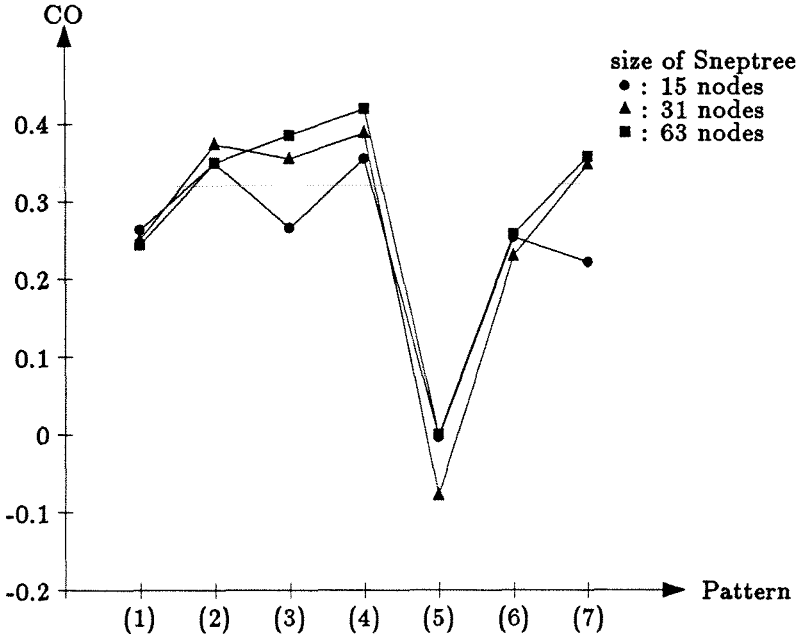

Figure B.12. The Average Communication Overhead CO of *sort*(N) with Algorithm V2

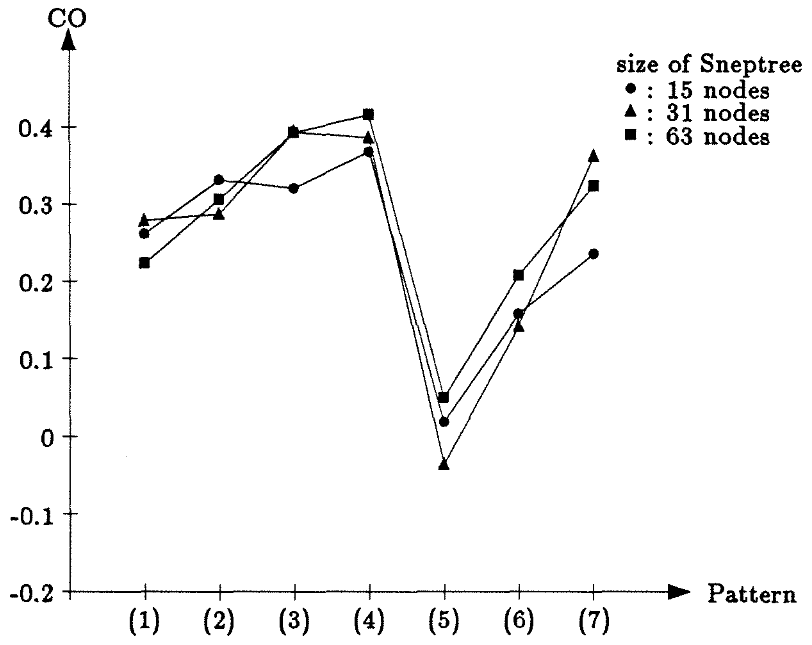

Figure B.13. The Average Communication Overhead CO of *sort*(N) with Algorithm V4

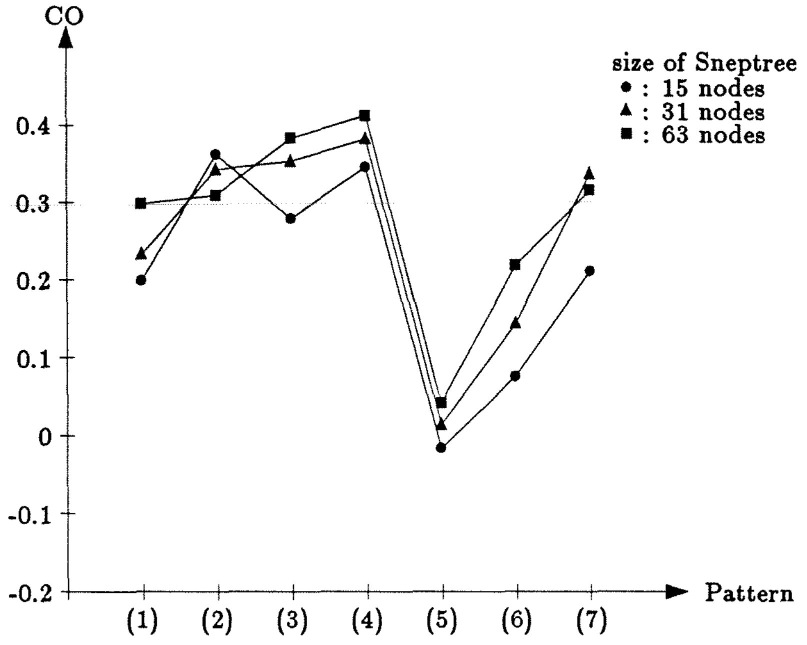

Figure B.14. The Average Communication Overhead CO of *sort*(N) with Algorithm V5

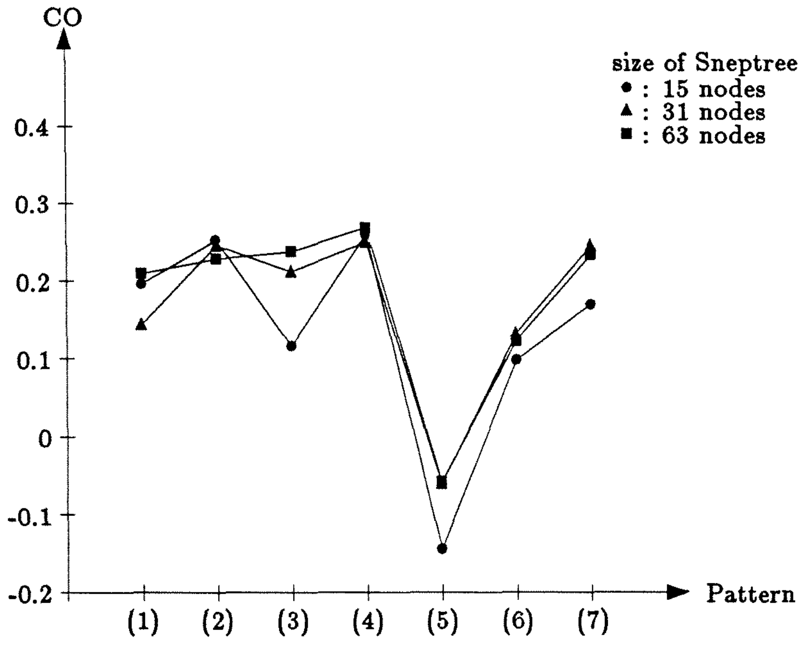

Figure B.15. The Average Communication Overhead CO of $a(N)$ with Algorithm V1

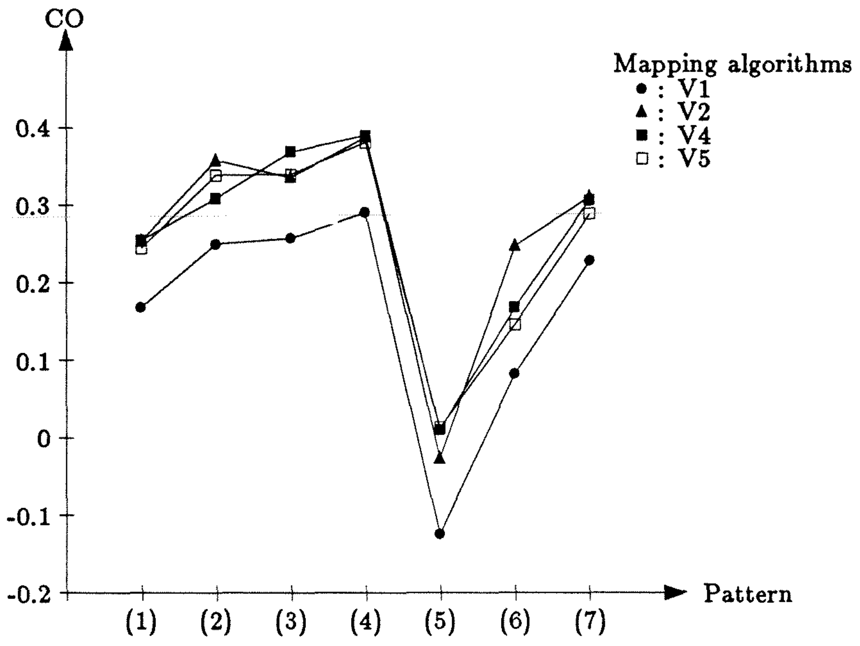

Figure B.16. The Average Communication Overhead CO of $a(N)$ for Different Algorithms