Dream Machine - A Platform for Efficient Implementation of Neural Networks with Arbitrarly Complex Interconnection Structures

Soheil Shams

CENG Technical Report 92-23

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4484

DREAM MACHINE - A PLATFORM FOR EFFICIENT IMPLEMENTATION OF NEURAL NETWORKS WITH ARBITRARLY COMPLEX INTERCONNECTION STRUCTURES

by

Soheil Shams

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

August 1992

Dedication

to my grandparents,

Davoud & Sima Shams

and

Nasrollah & Maryam Javanshir

Acknowledgments

I would like to thank Dr. Jean-Luc Gaudiot for serving as my advisor and for his encouragement and assistance in completing my Ph.D. studies. I would also like to thank Drs. Shahram Ghandeharizadeh, Keith Jenkins, and Viktor Prasanna for serving on my dissertation committee. A special thanks to Dr. Christof von der Malsburg for many hours of stimulating discussions during the early periods of my Ph.D. work. I am also grateful to Dr. Petar Simic for his collaboration on the optimization portion of my research. The numerous hours of discussion on optimization methods and parallel processing were one of the highlights of my research work. I would like to thank Scott Toborg for being a good friend and colleague. Many thanks to Dr. Wojtek Przytula for his support and collaboration. I would also like to thank Drs. Greg Nash and David Schwartz for their encouragement, support, and helpful comments on my research. I am grateful to the Hughes Aircraft Company for supporting my studies through their fellowship program. A special thanks to Barbara Dover in helping me with text editing and creating some of the figures in this dissertation.

Completion of this dissertation would not have been possible without the love and support of my family throughout all these years. I would like to specially thank my parents, Iraj and Sorour for always being there to give love, support, and encouragement. I like to also thank my brothers Fariborz and Sepehr for their love and support. Most of all, I would like to thank my love, Ladan, who in addition to helping me with many aspects of this dissertation, has been the main source of my energy in completing my degree.

Contents

Dedication ii

Acknowledgments iii

List of Figures ix

List of Tables xiii

Abstract xiv

1 Introduction 1

1.1 Problem Statement 1
1.1.1 The Architecture Design Problem 2
1.1.2 The Mapping Problem 3

1.2 Motivation 3
1.2.1 Why Parallel Implementations Are Necessary 4
1.2.2 Deficiencies of Current Implementations 4

1.3 Approach 5

1.4 Summary of Contributions 6

1.5 Organization of the Dissertation 7

2 Background 8

2.1 Neural Network Models 8
2.1.1 Neural Computation Overview 8
2.1.2 Representation of Neural Computation as Vector-Matrix Operations 10
2.1.3 Neural Network Interconnection Structures 11
2.1.4 Implementing Neural Network Learning Algorithms 13

2.2 Inherent Parallelisms of Neural Networks 14
2.2.1 Network Level Parallelism 15
2.2.2 Layer Level Parallelism 16
2.2.3 Neuron Level Parallelism 17
2.2.4 Synapse Level Parallelism 19

2.3 Parallel Implementation Methods 20
2.3.1 Implementations Applicable to Dense and Regularly Interconnected Neural Networks 22
2.3.2 Implementations Applicable to Sparsely Interconnected Neural Networks 24
2.3.3 Implementations Applicable to Arbitrarily Interconnected Neural Networks 26

3 The DREAM Machine 29

3.1 System Level Overview 30
3.1.1 System Organization 30
3.1.2 Execution Paradigm 31

3.2 Processing Element Design Description 32
3.2.1 Internals of the Processing Elements 33
3.2.2 Processor-Memory Interface 33
3.2.3 Instruction-Word Description 35

3.3 Implementing a Table Lookup Mechanism on the DREAM Machine 35

3.4 Interprocessor Communication Network 36
3.4.1 Nearest Neighbor Communications 36
3.4.2 Global Communications 38

4 Mapping Method Preliminaries 40

4.1 Mapping Principles 40
4.1.1 The General Mapping Method 41
4.1.2 The Algorithmic and Optimized Mapping Approaches 42

4.2 Mapping Neural Networks onto Fixed-Size Ring-Connected Architectures 43
4.2.1 System Utilization Characteristic of the Mapping Onto the Fixed-Size Ring Architecture 45
4.1.2 Execution Rate Characteristics of the Mapping Onto the Fixed-Size Ring Architecture 47

4.2.3 Mapping Multilayer Neural Networks Onto the Fixed-Size Ring Architecture 48
4.2.4 Deficiencies of the Mapping Onto the Fixed-Size Ring Architecture 48

4.3 Initial Mapping Method for 2-D Connected Architectures 49
4.3.1 Computational Aspects of the Back-Propagation Algorithm 51
4.3.2 Mapping Details 53
4.3.3 Implementing Neural Networks Larger than the Processor Array Size 59
4.3.4 Implementing Nettalk on the Hughes SCAP Architecture 61
4.3.5 Performance Evaluation of the Mapping Method 63
4.3.6 Applicability of the Mapping Method 64
4.3.7 Deficiencies and Shortcomings of the Mapping Method 65

5 The Algorithmic Mapping Method 66

5.1 Applicability of the Algorithmic Method 66

5.2 Implementation of Variable-Size Processing Rings on the DREAM Machine 67

5.3 Using Variable Length Rings to Process a Single Layer 68

5.4 Implementing Multilayer Neural Networks 71

5.5 Implementing Back-Propagation Algorithms 74

5.6 Implementing Block Connected Neural Networks 75
5.6.1 Implementing Data-Fusion Style Neural Networks 77
5.6.2 Implementing Feature-Detection Style Neural Networks 78

5.7 Implementing Neural Networks Larger than the Processor Array 78

5.8 Batch-Mode Implementation 79

5.9 Implementing Competitive Learning 80

5.10 Implementation Examples and Performance Evaluation 82
5.10.1 Performance Metric 82
5.10.2 Implementing a Fully Connected Multilayer Neural Network 83
5.10.3 Implementing a Block-Connected Multilayer Neural Network 84
5.10.4 Implementing a Fully Connected Single Layer Network 86

5.11 Analysis of Results 87

6 Optimization Based Mapping Method 90

6.1 Problem Overview 90
6.1.1 The Mapping Problem 91
6.1.2 Complexity of the Mapping Problem 91
6.1.3 Analogies to Other Combinatorial Optimization Problems 92

6.2 General Approach 93
6.2.1 Mapping Parallel Algorithms Onto Parallel Architectures 93
6.2.2 Addressing the Clustering Problem 94

6.3 Solving the Assignment Problem 95
6.3.1 Problem Representation 96
6.3.2 Assignment Cost Function Formulation 97
6.3.3 Constraints on the Assignment Matrix 98
6.3.4 The Complete Assignment Cost Function 100
6.3.5 Assignment of Neurons to Processors 101

6.4 Solving the Scheduling Problem 102
6.4.1 General Approach 102
6.4.2 Specific Problem Representation 104
6.4.3 Scheduling Cost Function Formulation 106
6.4.4 Constraints Terms of the Scheduling Cost Function 111
6.4.4.1 Architecture Dependent Constraints 111
6.4.4.2 Constraints on Path Variables 112
6.4.5 The Complete Scheduling Cost Function 113
6.4.5 Use of the Scheduling Procedure for Mapping Neural Networks 115

6.5 Constraint Nets Used for Optimization 117

6.6 Optimization Procedure 120
6.6.1 Update Equations for the Assignment Problem 121
6.6.2 Update Equations for the Scheduling Problem 122
6.6.3 Implementing the Optimization Procedure 127

7 Results of the Optimization Procedure 128

7.1 Implementation of the Optimization Algorithm 128
7.1.1 Control File Format 128
7.1.2 The Assignment and Scheduling Optimization Routines 130

7.2 Results of the Assignment Procedure 132
7.2.1 Results from Implementing the Bokhari Example 132
7.2.1.1 Problem Description 133
7.2.2 Results from Implementing the Everstine Example 142

7.3 Results of the Scheduling Procedure 144
7.3.1 Receptive-Field Example 144
7.3.2 Randomly Sparse Matrix Example 150

8 Asymptotic Performance Analysis 153

8.1 Implementation Complexity of Neural Network Processing 153
8.1.1 Mapping Method Comparisons 154
8.1.2 Area-Time Complexity of Neural Network Implementation Methods 156
8.1.2.1 Complexity of the Fixed-Size Ring Implementation 156
8.1.2.2 Complexity of the Variable-Length Ring Implementation 157
8.1.2.3 Complexity of the Optimization Based Implementation 158

8.2 Complexity Analysis of the Optimization Based Mapping Method 159
8.2.1 Time Complexity of the Assignment Problem 159
8.2.2 Time Complexity of the Scheduling Problem 160
8.2.3 Tradeoffs on the Use of the Optimization Procedure 161

8.3 Performance Comparison 162

9 Conclusion 165

9.1 Summary 165

9.2 Future Research Directions 167

References 170

List of Figures

Figure 1-1 - Basic structure of a neural network 2

Figure 2-1 - Computation performed by neuron *i*. 9

Figure 2-2 - Typical transfer functions used by neural networks. (a) Threshold function. (b) Ramp function. (c) Sigmoid function 11

Figure 2-3 - Examples of neural network interconnection structures. (a) One layer fully connected network. (b) Two layer network. (c) Two layer network with limited receptive field interconnections. 13

Figure 2-4 - Exploiting network level parallelism by mapping complete networks to individual PEs. 15

Figure 2-5 - Exploiting layer level parallelism by mapping consecutive layers of the neural network to consecutive PEs of the processing pipeline 16

Figure 2-6 - Exploiting neuron level parallelism by mapping individual neurons to distinct PEs 18

Figure 2-7 - Exploiting synapse level parallelism by mapping single synapses to each PE. In this figure s denotes the number of synaptic connections used in the neural network 19

Figure 2-8 - Implementing a fully connected neural network with 16 neurons on a ring-connected systolic array. (a) Assignment of neurons to processors and flow of data. (b) Computation performed in each PE at time t 23

Figure 2-9 - Using network level parallelism for implementing neural networks on the Warp machine. 27

Figure 2-10 - Implementation of neural networks on the connection machine, from [79]. 27

Figure 3-1 - The top level design of the DREAM Machine 30

Figure 3-2 - Top level design of the Processing Elements. 32

Figure 3-3 - Processor and memory detail diagram. Associated with each data value in the local memory of each PE is a switch setting value used to configure the communication topology of the machine. 34

Figure 3-4 - A single processor and its associated interprocessor communication switches. 37

Figure 3-5 - Schematic of the reconfigurable interprocessor communication switch..... 38

Figure 4-1 - Mapping neurons to PEs and constructing a computation path between the PEs. 42

Figure 4-2 - Mapping a fully connected neural network onto a ring systolic processing array. 44

Figure 4-3 - Exploiting network level and neuron level parallelism on a 2-D mesh connected systolic architecture..... 50

Figure 4-4 - Recall phase (forward pass) processing of neuron i on layer ℓ of the multilayer perceptron neural network. 51

Figure 4-5 - Learning phase (error back-propagation) processing for neuron i on layer ℓ of the back-propagation algorithm..... 52

Figure 4-6 - Data organization in the processor array during the first cycle of the recall phase. Only processors in the first row are enabled..... 53

Figure 4-7 - Processing in each processor of the array during the recall phase..... 54

Figure 4-8 - Next two cycles of the data flow through the array after initial cycle shown in Figure 4-6. 55

Figure 4-9 - Data organization in the processor array during the first cycle of the learning phase. Only processors in the first row are enabled. 56

Figure 4-10 - Processing in each enabled processor of the array during the learning phase..... 57

Figure 4-11 - Next two cycles of the data flow through the array after initial cycle shown in Figure 4-9. Shaded processors are disabled from performing calculations..... 58

Figure 4-12 - Network partitioning for implementation on a "3 x 3" processor array..... 59

Figure 4-13 - System architecture of the Hughes Systolic/Cellular Coprocessor..... 61

Figure 5-1 - Using the reconfigurable switches of the DREAM Machine to construct circular rings on the processing array. (a) A single ring of size 59. (b) Three disjoint rings of different sizes executed in parallel..... 68

Figure 5-2 - Plot of R_t vs. N_t with P=256 processors..... 70

Figure 5-3 - Plot of T_t vs. N_t for the fixed-size ring mapping defined by equation (4-5), and for the variable-size ring mapping defined by equation (5-3). The parameters $k1$ and $k2$ are assumed to be 1 in order to simplify the comparison..... 71

Figure 5-4 - A three layer neural network with $N_1>N_2>N_3$. 72

Figure 5-5 - An embedded ring structure containing a ring of size N_2 embedded in another ring of size N_1 73

Figure 5-6 - A blocked structured neural network utilizing 3 disjoint blocks between the input and hidden layers and its associated mapping on 3 processing rings 76

Figure 5-7 - Memory locations of synaptic weights on the inputs of neuron i..... 81

Figure 5-8 - A neural network structure for image compression and decompression with a regularly blocked structure [56]..... 85

Figure 5-9 - The ring structure associated with the image compression and decompression network of Figure 5-8..... 86

Figure 6-1 - Mapping of the nodes of a process graph **D** onto a processor graph **G** using the assignment matrix $\boldsymbol{\omega}$ 96

Figure 6-3 - An example data dependence graph for a simple arithmetic calculation 103

Figure 6-4 - Several different dependence graphs performing the same computation due to the commutative nature of the addition operation..... 104

Figure 6-5 - A 3-D representation of 3 non-intersecting path traversals in time. The left hand plane represents the 2-D processor array..... 105

Figure 6-6 - An example of a path traversal and its associated penalty term contributions due to repeated crossings of a neuron m required for its computation. The value of $\eta^{i}_{m(\alpha)}$ can be any value in the range [0,1] in the shaded region. 110

Figure 7-1 - Interconnection topology of a 6x6 Finite Element Machine (FEM). 133

Figure 7-2 - The interconnection matrix G^{AB} describing the 6x6 FEM topology..... 134

Figure 7-3 - A 33 node finite element structure used as the process graph. 135

Figure 7-4 - The process graph interconnection matrix $D^{\alpha n}$ 136

Figure 7-5 - Interprocessor communication distance matrix $\mathbb{G}^{AB}$ obtained from the architecture topology graph G^{AB}. 137

Figure 7-6 - The Histogram associated with 1000 randomly generated assignments for the 33 node Bokhari example..... 138

Figure 7-7 - Histogram of solution cardinalities for the 1231 runs described in Table 7-3. The global optimum solution is at cardinality 78 which is found the largest percentage of times. 139

Figure 7-8 - Graph of system temperature and cardinality value vs. sweep number..... 140

Figure 7-9 - Graph of system temperature, total cost (given by equation (6-13)), and mismatch cost (given by equation (6-1)) vs. sweep number..... 140

Figure 7-10 - Graph of system temperature and the various penalty terms (given by equations (6-6), (6-7), (6-11), and (6-12)) vs. sweep number..... 141

Figure 7-11 - Histogram of the solution cardinalities when mapping the 25 node process graph to FEM of size 5x5 taken from a collection of 736 runs..... 141

Figure 7-12 - The histogram associated with solution cardinalities of 1000 random assignments of the Everstine 59 node example mapped to a 4-nearest neighbor connected 8x8 parallel processing architecture..... 143

Figure 7-13 - Neural network with a receptive field type interconnection structure..... 145

Figure 7-14 - Scheduling flow pattern for the 4 paths associated with Figure 7-13. The shaded boxes indicate the required neurons to be traversed by each path..... 146

Figure 7-15 - A second scheduling flow pattern for the 4 paths of Figure 7-13, with path length M=11. The shaded boxes indicate the required neurons to be traversed by each path. A loop in the path indicates that the path stayed in the processor for the corresponding cycle..... 148

Figure 7-15 - Scheduling flow pattern for the 4 paths of Figure 7-13, with optimum path length (M=9)..... 149

Figure 7-16 - Interconnection matrix associated with a sparse iterative system..... 150

Figure 7-17 - Four of the 16 paths associated with the sparse iterative system shown in Figure 7-16. The shaded boxes indicate neurons which must be traversed by the associated paths. The boxes with thick outlines indicate start and end processors of each path..... 151

List of Tables

Table 4-1 - Comparison of different implementations of the Nettalk neural network based on throughput and local memory size requirements..... 64

Table 5-1 - Performance comparison of the variable-size ring mapping vs. the fixed-size ring mapping based on the MCPS metric. 88

Table 5-2 - Performance comparison of the variable-size ring mapping vs. the fixed-size ring mapping based on the optimality ratio..... 89

Table 7-1 - Control file format used for the assignment routine 129

Table 7-2 - Control file format used for the scheduling routine..... 130

Table 7-3 - Statistic collected from executing the assignment optimization routine on the 33 node Bokhari example. The shaded row is the results of a Monte-Carlo search algorithm used for comparison. 137

Table 7-4 - Statistics collected from executing the assignment optimization routine on the 59 node Everstine example. The shaded row is the results of a Monte-Carlo search algorithm used for comparison. 144

Table 8-1 - Comparison of several implementation methods used for neural network processing. 163

Table 8-2 - Comparison of several implementation methods used for neural network processing, assuming a constant number of connections per neuron..... 164

Abstract

In this dissertation, the Dynamically Reconfigurable Extended Array Multiprocessor (DREAM) Machine is presented for efficient implementation of neural network models. we identify several sources of parallelism, inherently available in neural network models. We show that the Single Instruction stream Multiple Data stream (SIMD) execution paradigm can be an effective and efficient method for parallel implementations of neural networks. We argue that the current implementation methods are only applicable to neural networks with specific characteristics in their interconnection structures, such as being dense and regularly interconnected. In light of this argument, we present a general processing scheme for implementing neural network models on systolic parallel computers. This processing scheme is realized using two different mapping methods which require a limited degree of communication autonomy at each processor, as offered by the DREAM Machine architecture. The first is an algorithmic method based on constructing variable-length processing rings on the DREAM Machine architecture. The second method involves an optimization procedure in which efficient mappings of neural networks with arbitrary interconnections onto the DREAM Machine are automatically generated. We show, both analytically and empirically, that these mapping methods allow the DREAM Machine to be an effective architecture for implementing neural network models with a variety of different interconnection structures.

Chapter 1

Introduction

Performing complex numerical calculations, such as evaluating π to the 100^{th} significant digit, could take a human several months or even years, where as a simple digital computer can perform the task in a few seconds. On the other hand, recognizing a familiar face in a crowd is a task that a human can perform rather effortlessly but it cannot be performed effectively even with the most powerful computers. This kind of problem is at the center of the current surge in research in neural networks. Neural network models are intended to solve problems that are readily performed by humans (e.g., vision and speech), but have been difficult to implement using conventional algorithms. This dissertation presents the Dynamically Reconfigurable Extended Array Multiprocessor (DREAM) Machine as a platform for efficiently implementing the computation associated with processing a wide range of neural network models.

1.1 Problem Statement

As the term "neural network" implies, these models are based, to varying degrees, on the biological nervous system. A great deal of effort has been put into developing a general model for neural computation. Unfortunately, no single model, or even a class of models, have proven to be effective in addressing the broad range of "human-like" abilities. This has resulted in a lack of a clear-cut definition of what constitutes a neural network model. The fundamental principles that make up a substantial number of current neural network models, and are expected to be consistent with further developments in neural modeling research, can be described as follows. Neural network models consist

of two major components: processing units (referred to as neurons) and connections (referred to as synapses). The synapses are used for transferring and possibly modifying the information transferred between neurons, see Figure 1-1.

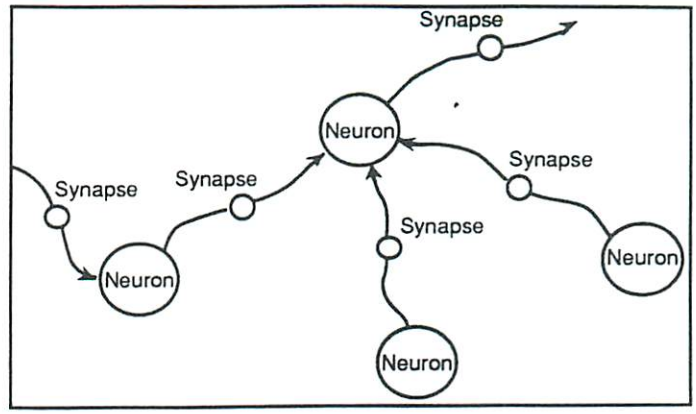


Figure 1-1 - Basic structure of a neural network

The problem being addressed in this dissertation is how to implement these models so that the required computation can be completed in the shortest amount of time. An obvious solution is to use the inherent parallelism available in neural networks to perform multiple operations in parallel. This solution, in turn, poses two major questions: what type of parallel processing architecture is best suited for neural network processing, and how to map the computation associated with neural networks onto a parallel processing architecture. These two problems are the primary issues being addressed in this dissertation.

1.1.1 The Architecture Design Problem

The difficulty in solving the implementation problems lays in the fact that neural network models cover a wide range of algorithms, each having a specific computational requirements. In addition to the differences in the computational requirements among various neural networks, the significant diversity in the complex interconnection structures of these models, even between different applications of the same model, makes efficient general-purpose implementations difficult to realize. There is an inherent trade-off between execution speed and the degree of programmability offered by an implementation. High speed implementations can be developed which take advantage of all the available peculiarities of a particular neural network to optimize the design of the architecture. Such an implementation would be of limited use, since it can only be used

for implementing a single neural network. On the other hand, allowing for flexibility in the implementation, reduces its execution speed. Achieving a good compromise between flexibility and speed, in implementing neural network models, is a prime objective of this dissertation.

1.1.2 The Mapping Problem

A major issue in efficient utilization of any parallel processing architecture involves the mapping of processing modules of a specific computational task to individual processors of the architecture in an optimal manner. The optimality criteria could be based on various parameters, such as execution rate, memory utilization, and interprocessor communications. The problem of mapping neural network models on parallel architectures involves three basic operations. First, we must identify the individual processes involved in the computation of neural networks which can be executed in parallel. The goal here is to identify those processes which require minimal amount of communications and are sufficiently fundamental to the computation as to be applicable to a wide range of models. Second part of the mapping problem is to arrive at an efficient method for assigning each process to a unique processor in order to achieve efficient processing. Finally, a schedule for the sequence of computation and communication operations, associated with the implementation of the neural network model, must be generated.

1.2 Motivation

The computational paradigm used by neural networks is quite different than that used by conventional algorithms. A conventional algorithm specifies the exact order and the specific operation to be performed at each time step in order to implement a given task. A neural network, on the other hand, "computes" based on the specific characteristics of the neurons and the specific interconnection structure of the network. This fact highlights a major difference between the method used for programming neural networks as compared to that used for a conventional algorithms. This rather radical method of programming forces an application developer to think in terms of simple processing units and their interactions as opposed to a specific flow of operations.

Numerous advantages have been cited for applying neural network models to complex problems that do not lend themselves well to procedural implementations [1, 26]. These advantages include: the ability of neural networks to be used as robust pattern associators or pattern classifiers, the ability to self-organize or learn with or without explicit information from an outside supervisor, the ability to generalize based on previously learned information, and inherent fault-tolerance achieved by using distributed representation and processing. In addition, an attractive attribute of neural networks that has generated much enthusiasm for expecting high implementation speeds is associated with their inherently parallel computational structure. It is desirable to efficiently harness this parallelism by implementing neural network algorithms on parallel processing computers.

1.2.1 Why Parallel Implementations Are Necessary

Clearly, neural networks employ a significantly different type of computational model than that used by conventional algorithms. This fact has generated a need for special architectural designs that can best support distributed processing required by such models. Although neural networks can solve computationally difficult problems, such as the Traveling Salesman Problem (TSP), in a relatively short number of iterations [13, 33], each iteration of a neural network consists of updating a large number of neurons and synaptic connections in parallel. Implementing these neural network models on conventional uniprocessors enforces serialization of an inherently parallel computation and thus will lead to low throughput rates. With the continual research in neural network model development, and their application to an ever increasing number of real-world problems, the number of neurons and interconnections being employed by these models is increasing rapidly. For these systems to be used in real-world applications, an efficient hardware platform is required to utilize the inherent parallelism available in these models to its fullest extend.

1.2.2 Deficiencies of Current Implementations

Although conventional parallel architectures can, to a certain degree, help in speeding up neural network processing, as described in Chapter 2, they lack many features required for efficient implementation of neural networks. Furthermore, many special-purpose

parallel machines lack the required flexibility needed to be applicable to the wide range of neural network applications currently being developed.

Two main factors have limited the amount of parallelism that can realistically and efficiently be utilized by parallel implementations. The first is due to the high communication requirements between individual neurons in the network. A neuron requires input from many other neurons during each processing cycle in order to produce an output value. If each neuron is assigned to a unique processing element (PE), high interprocessor communication bandwidth is required to accommodate this exchange of output values among the various neurons. The second problem facing parallel implementations of neural networks is attributed to the uncertainty in the specific neural computational requirements. Any practical implementation method should offer sufficient flexibility so that a wide array of neural networks with diverse interconnection structures and computational needs can be efficiently processed. Without such flexibility the applicability of the implementation and its useful life expectancy will be greatly limited.

1.3 Approach

The approach taken in this dissertation, at solving the problems outlined in Section 1.1, involves three parts. First, we organize and represent all neural computation using a uniform representation scheme. We determine the specific areas where parallelism can efficiently be exploited to attain maximum speedup. At the same time, we represent neural computation sufficiently general so that it may be applicable to a wide range of neural network models.

Second, we propose a special-purpose architecture which can be effectively utilized for neural network processing. The design of this architecture is accomplished by identifying the most compute intensive operations performed by neural network algorithms. This information can be used to guide the design of the architecture such that these operations are performed more efficiently. This may be accomplished through special-purpose hardware mechanisms, or through special mapping techniques.

Finally, we propose a mapping method for efficiently implementing various neural network models on this architecture. This mapping method is based on the fundamental operations performed by most neural network models. Therefore, it can be applied to a

large class of neural network models. In addition to implementing different neural network models, the implementation will be able to efficiently implement networks with arbitrary interconnection structures. The approach here is to introduce flexibility in the communication network of the architecture to be able to support irregular and dynamically changing flow patterns. Such a capability can be efficiently utilized to match the neural network interconnection structure to the topology of the computer architecture.

1.4 Summary of Contributions

The contributions made by the research reported in this dissertation are listed below:

- Various sources of parallelism available in neural network computation have been identified.
- A mapping method for implementing neural network computation on 2-D mesh-connected systolic arrays has been developed and demonstrated.
- The DREAM Machine, a reconfigurable parallel processing architecture with specialized features for efficient implementation of neural networks with arbitrary interconnection structures, has been proposed.
- An algorithmic mapping method has been introduced to efficiently implement neural networks with regular interconnection structures on the DREAM Machine.
- A general approach for mapping parallel algorithms onto parallel processing architectures, based on combinatorial optimization techniques has been developed.
- An optimization based mapping approach has been used to automatically generate efficient mappings of neural network structures onto the DREAM Machine architecture.

1.5 Organization of the Dissertation

In Chapter 2, we present background material relating to neural network computation and their associated parallel processing implementations. In this chapter we identify and present the various sources of parallelism available in neural computation. A taxonomy of various interconnection structures, employed by neural network models, is also presented in Chapter 2. In Chapter 3, we present the DREAM Machine architecture. The specific architectural features of this architecture, which have been specifically design for neural network processing, are described in detail. We show the basic mapping method used for implementing neural network models on parallel systolic architectures in Chapter 4. A detailed analysis of a mapping method, proposed for implementing neural networks on ring-connected systolic architectures, is performed in this chapter.

In Chapter 5, an algorithmic mapping method is introduced to implement regularly interconnected neural networks on the DREAM Machine. The effectiveness of this mapping algorithm is demonstrated through several examples. The performance of this method is analyzed and compared to that of the linear ring implementation. An analytical measure for evaluating the *goodness* of a particular mapping is presented in Chapter 6. This is done by deriving two cost functions, one associated with the problem of assigning neurons to process, and another associated with generating the necessary scheduling of operations required for implementation of a particular neural network. The basic concepts of Constraint nets (Cnet) [78] used for solving combinatorial optimization problems, are given in Chapter 6. The use of the Cnet method for solving the assignment and scheduling optimization problems is shown in Chapter 6.

Empirical results of implementing the optimization based mapping on several benchmark examples are given in Chapter 7. The asymptotic performance of our implementation method, along with a time complexity of our optimization based mapping procedure, are analyzed and compared to other methods in Chapter 8. The conclusions are gathered in Chapter 9.

Chapter 2

Background

In this chapter we will describe neural network processing viewed from a computation/communication requirements perspective. This approach enables us to effectively highlight various inherently parallel structures available in neural networks. We will outline several levels of parallelism that can be used for efficient implementations on parallel architectures, and will demonstrate their usage through several examples.

2.1 Neural Network Models

Neural networks cover a wide spectrum of models with varying computational and structural characteristics. In order to design a general-purpose neurocomputer which can efficiently implement a large number of these models, fundamental operations which are common to all neural computations must be identified. Below, we describe the principle computations performed by neural network models and define a taxonomy of neural network interconnection structures.

2.1.1 Neural Computation Overview

As described in the introductory chapter, neural network models are comprised of two basic components, neurons and synapses. Neurons perform simple computations while synapses communicate the result of these computations among the neurons. A general formula for the computation performed by a neuron can be stated as

$$a_i = f(u_i + \theta_i), \qquad \text{where} \tag{2-1}$$

$$u_i = \sum_{j=1}^{N} w_{ij} a_j \qquad \text{and} \tag{2-2}$$

a_i is the neuron output value, θ_i is the neuron threshold value, w_{ij} is the synaptic connection weight between neuron i and neuron j, and f is the neuron transfer function. This computation is graphically depicted in Figure 2-1. Neural computation is comprised of two classes of operations: local, and distributed. Local operations are those which require data values locally available to each neuron, such as the application of the transfer function f, the addition of the threshold value θ, and other model specific operations involving local parameter values. On the other hand, distributed operations require the transfer of neuron output values among the neurons in the network by means of the synaptic interconnection network. The calculation of the weighted sum of neuron output values received by a particular neuron, designated by the summation operator in equation (2-2), is an example of a distributed operation.

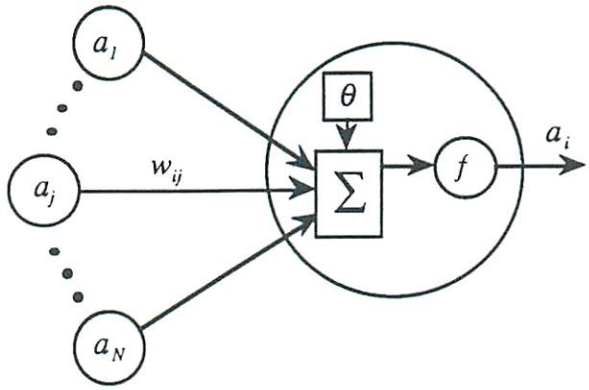


Figure 2-1 - Computation performed by neuron i.

In addition to the neural computation described by equation (2-1), neural network models employ various learning algorithms. Learning algorithms are used to modify the synaptic interconnection weights, and possibly the neuron threshold values, in order to produce a desired response from the network. The computational requirements of these algorithms can also be divided into local and distributed operations. Similar to the case of neural computation, the distributed operations of the learning algorithm are of greatest interest to us as they require communication among parallel operations performed by neurons. In this section we will describe computational characteristics of a few common neural network learning algorithms.

2.1.2 Representation of Neural Computation as Vector-Matrix Operations

Vector and matrix notation can be utilized in order to obtain a systematic and uniform representation of neural network computation. Neural network models and various learning algorithms can be formulated using vector-matrix operations [42]. In such a scheme a vector **a** is used to represent the state of the neurons in the network and a matrix **W** is used to denote the synaptic interconnection network. Equation (2-1) can be rewritten using this notation as a matrix-vector-product operation

$$\mathbf{a}^{(t+1)} = f\left(\mathbf{W}\mathbf{a}^{(t)}\right), \tag{2-3}$$

where the superscript *(t)* denotes the iteration index. In such a representation, the threshold value associated with each neuron can be implemented as an additional synaptic weight received from a neuron with a constant output value of one. A great many neural network models can be represented in this form depending on the choice of the iteration index *(t)*, the activation function *f*, and the specific restrictions imposed on the interconnection structure **W** [42]. Similar approaches involving vector-matrix-product and inner-product operations can be used to implement various learning algorithms.

It can be observed from equation (2-1) that the two primary operations performed by neural networks are: the application of the transfer function *f*, and the calculation of the weighted input sum to a neuron. The application of the transfer function is most often not dependent on the interconnection structure of the network. Examples of some common transfer functions used by neural networks are the threshold function, the ramp function, and the sigmoid function (see Figure 2-2). All of these functions can be implemented locally by each neuron. A widely used approach for fast evaluation of complex transfer functions (such as the sigmoid) is through the use of a table lookup mechanism. Depending on the amount of quantization tolerated by the neural network model, an appropriately sized portion of memory is allocated as a lookup table which each neuron uses to determine its output value.

From a parallel processing point of view, the min/max function is a more challenging type of transfer function to be implemented. The min/max function is used to identify an output neuron with either the largest or the smallest output value depending on the neural network model. These functions are used by competitive models employing a "winner-take-all" scheme, such as the Bidirectional Associative Memory (BAM) [38] and

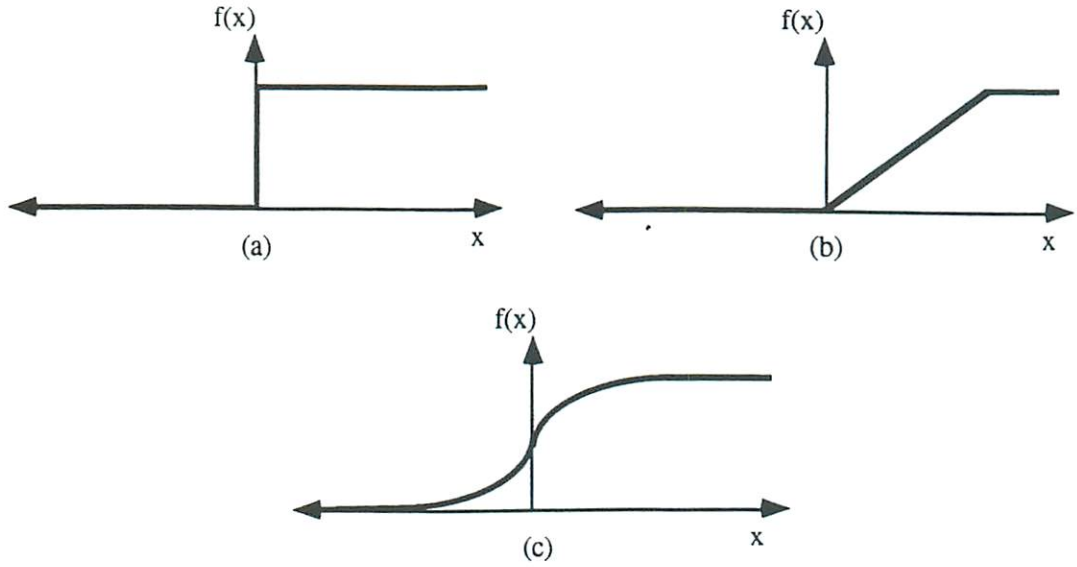


Figure 2-2 - Typical transfer functions used by neural networks. (a) Threshold function. (b) Ramp function. (c) Sigmoid function.

the Adaptive Resonance Theory (ART) [10]. A min/max transfer function cannot be implemented based solely on the local information in each neuron since the final state of the network is a function of all the neurons competing with each other. Support for parallel implementation of such functions has been rather limited and will be discussed in more detail later in this chapter. A method for efficiently supporting min/max type operations on the DREAM Machine is presented in Chapter 3.

2.1.3 Neural Network Interconnection Structures

A major obstacle in devising a general method for efficient implementation of neural networks on parallel architectures is associated with the complex and diverse range of neural network interconnection structures. A general taxonomy of interconnection structures is presented here in order to systematically analyze various methods for their realization on parallel hardware. First, let us define some commonly used terminology. A *layer* refers to a set of neurons that are only connected to the neurons of the adjacent layers. An example of a two layer network is depicted in Figure 2-3(b). It should be noted here that others might refer to such a structure (Figure 2-3(b)) as a one layer network by counting the layers of synapses rather than neurons.

The simplest interconnection structure is one in which every neuron is connected to all the other neurons in the network, see Figure 2-3(a). Such an interconnection structure can be represented as a matrix **W** with all non-zero off-diagonal elements. Generally, such networks perform iterative update calculation with index *(t)* of equation (3) denoting time. An example of such a model with a dense and symmetric interconnection structure is the Hopfield network [32]. An extension of this interconnection structure is used in two layer iterative models (see Figure 2-3(b)) such as the Bidirectional Associative Memory (BAM) [38] and the Adaptive Resonance Theory (ART) [10] models. In these models, information flows between one layer to another and back until the system converges to a stable state. Therefore, the iteration index *(t)* of equation (2-3) refers to an alternating layer index switching between the first and second layers.

However, a number of two layer neural network models do not iterate between the layers, rather input data is presented to the first layer and the network produces a corresponding response in the second layer. The Kohonen's self-organizing feature maps [37] and the Perceptron [65] are examples of such models. This type of layered neural network models can be extended to include models with multiple layers of neurons. The widely used Multilayer Perceptron [66] and the neocognitron [19] are examples of models with such multilayer interconnection structures. These layered neural networks employ "feed-forward" processing. That is, the iteration index of equation (2-3) is used as a layer index ranging from the input layer number to the output layer number.

In the discussion of various interconnection structures above, we did not consider the interconnection density and possible restrictions on the synaptic weight values imposed by the neural network model. Most neural networks have been formulated with full interconnections between neurons of adjacent layers. Several weight pruning techniques have been proposed to eliminate a large number of these connections to improve the generalization characteristics of the model [46]. As neural network models continue to be applied to real-world applications, a priori knowledge about the problem will be increasingly used to eliminate a great many interconnections, resulting in a more sparse and better structured interconnection network, such as the one shown in Figure 2-3(c). Furthermore, a priori knowledge about the problem can enforce certain restrictions on the weight values in the interconnection network. For example, the neocognitron network contains many feature sensitive neurons. These neurons have a limited number of

connections to the previous layer such that each neuron is connected to only a small neighborhood (or receptive field) of neurons in the previous layer. The model also requires that all the neurons associated with a specific feature type to have the same weight value distribution. This method has also been termed weight sharing since a single weight value is shared among a number of different neuron pairs. Each feature detecting neuron can thus be tuned to have a maximum output value when a particular activity pattern is presented in its input receptive field area.

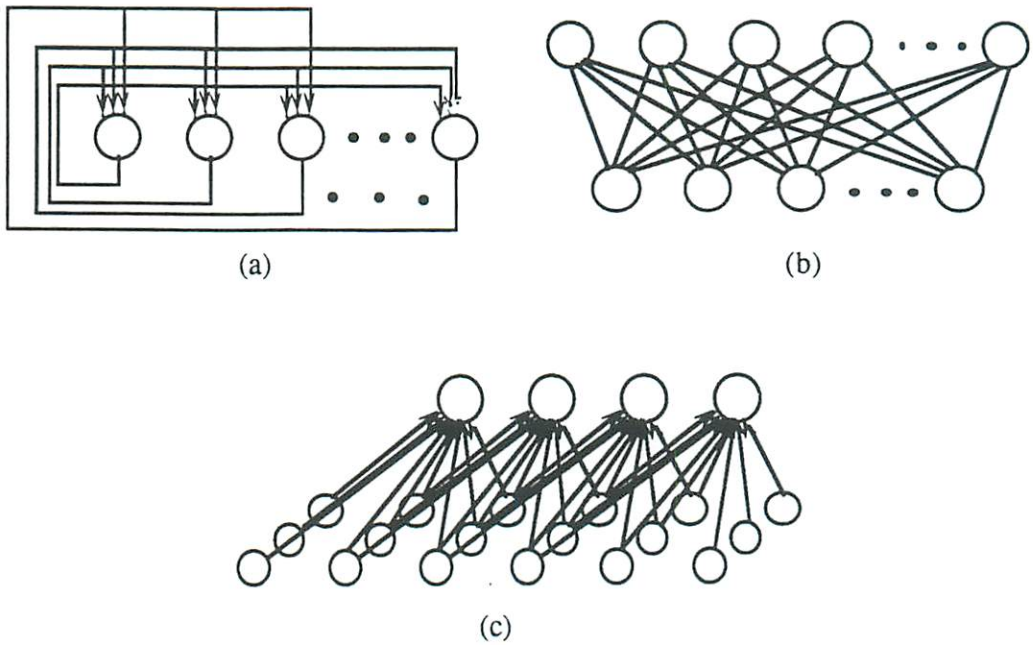


Figure 2-3 - Examples of neural network interconnection structures. (a) One layer fully connected network. (b) Two layer network. (c) Two layer network with limited receptive field interconnections.

2.1.4 Implementing Neural Network Learning Algorithms

In addition to parallel implementations appropriate for neural computation, we are also interested in methods for efficient implementation of learning algorithms. Neural network learning can be classified into three classes. The first is called *one-time* learning. In this method, the synaptic interconnection weights are determined without the use of neural computation and are based only on a specific equation (as in the case of the Hopfield network [33]) or on the actual patterns to be memorized (as in the case of the BAM [38]

and the PNN [80] models). The second class of learning algorithms is called *unsupervised*. Models utilizing unsupervised learning generate appropriate weight modification values based on the input pattern received and the current state of the network. Unsupervised learning algorithms generally employ a competitive layer, as described in Section 2.1.2, where a number of neurons compete to find the maximum or minimum output activation value. The third type of learning algorithm is the *supervised* learning method. This method requires an outside teacher to advise the network of the correct response for a given input pattern. This information is consequently used by the neural network to generate appropriate weight modification values.

From a computational point of view, one-time learning algorithms can be implemented "off-line" using any available method, since the calculation is performed only once and does not cause a computational bottle-neck. The unsupervised learning strategies employing competitive learning, such as the Kohonen's SFM [37], require a method for selectively modifying synaptic weights associated with the winning neuron and possibly a number of other neurons in its local neighborhood. Since only the weights associated with one or a few neurons in the network are updated, the amount of available parallelism of such algorithms is fairly limited. Supervised learning methods, such as the back-propagation algorithm [66], generally propagate error values through the neural network using the synaptic interconnections. In most instances, an efficient implementation for forward propagation of data, used for neural computation, can efficiently be applied in the learning phase.

2.2 Inherent Parallelisms of Neural Networks

In this section, we will describe four types of parallelism, inherently available in neural networks, that can be exploited by implementations on parallel processing architectures. Each type of parallelism corresponds to a different level of computational granularity. These levels are designated here, in order of decreasing granularity, as network level, layer level, neuron level, and synapse level parallelism. Mapping methods can utilize one or more of these parallel computational structures to assign neural network computation to processors in a parallel processing architecture. In the following paragraphs we will describe in detail the various mapping methods and corresponding

computer architectures used for efficient implementations. This description is organized based on the level of parallelism used by each method.

2.2.1 Network Level Parallelism

Network level parallelism involves mapping one complete neural network to each processing element (PE) of a parallel processing system, see Figure 2-4. Employing this type of parallelism increases the total throughput of the machine by executing a number of networks in parallel, but the execution rate for a single network is not increased. Network level parallelism can be used efficiently for various applications. For example, several neural network models use a number of parameters that are determined empirically through a trail-and-error procedure. In a machine with P processing elements, P different parameter settings can be evaluated in parallel by executing one neural network per processor, using network level parallelism. The solution of many iterative neural network models, such as the Hopfield net [33], Constrained net [78], and Elastic net [13], used in optimization applications is dependent on the initial state of the network. Identical networks with different initial states can be implemented on a parallel processing architecture utilizing network level parallelism. This type of mapping usually leads to a linear speedup factor since there is no need for communication between processors of the architecture.

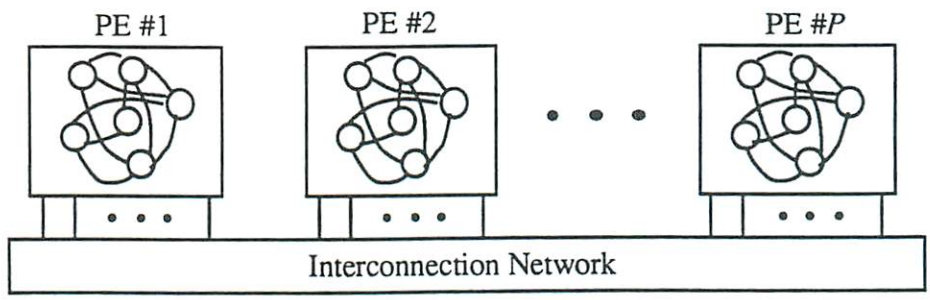


Figure 2-4 - Exploiting network level parallelism by mapping complete networks to individual PEs.

Network level parallelism can also be used to increase the throughput of some neural network learning algorithms. In particular, learning algorithms which allow for the so-called batch-mode training, such as the back-propagation algorithm, modify the network's synaptic weights after summing individual weight contribution of a number of different training patterns. Parallel implementation of this technique can be accomplished

by assigning a different input pattern to each of the identical neural networks implemented in different processors. The weight change contribution of individual input patterns are summed together across the interprocessor communication network to arrive at the final weight modification values. Each processor in the system uses the summed modification values to update its copy of the network. Unlike the previously described uses of network level parallelism, this approach requires communication of weight modification values among all the processors in the machine. Therefore, efficient utilization of this approach requires high bandwidth communications between the PEs of the target architecture. An example of such an implementation is presented later in this chapter. Network level parallelism in general is applicable to coarse grain parallel processing architectures that offer powerful processing elements with a large amount of memory.

2.2.2 Layer Level Parallelism

Layer level parallelism involves mapping each layer of a neural network to individual processors of the architecture, see Figure 2-5. It is obvious that this type of parallelism is applicable to layered neural network models. Furthermore, the amount of parallelism gained through the use of this method is directly related to the number of layers in the network. Since most neural network models contain rather a few (less than 10) layers, parallel implementations exploiting only layer level parallelism offer very limited speedup over uniprocessor implementations. Therefore, implementation methods using layer level parallelism are often utilized in conjunction with one or more of the other type of parallelisms.

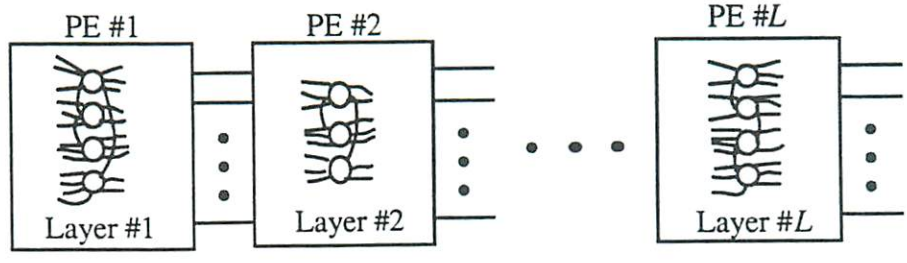


Figure 2-5 - Exploiting layer level parallelism by mapping consecutive layers of the neural network to consecutive PEs of the processing pipeline.

When implementing non-iterative feed-forward neural network models, such as the Multilayer Perceptron (MLP) network [66], the iteration index *(t)* of equation (2-3) is

used as an index over the layer numbers; thus the computation is performed sequentially from one layer to the next. In order to take advantage of layer level parallelism, it is necessary to process a number of different input patterns concurrently in a pipelined fashion. In effect, layer level parallelism implements L complete networks concurrently, where L is the number of layers in the network. The restrictions regarding the applicability of this method for implementing neural networks are similar to those imposed by network level parallelism. The major difference here is that the effective throughput of the system is equal to the time associated with processing of the largest layer in the network, where as the throughput of implementations utilizing network level parallelism is equal to the time required to process a complete network.

As defined in Section 2.1.3, the neurons of a given layer, in a multilayer neural network, are only connected to neurons of the adjacent layers and possibly to each other. This type of interconnection structure requires the computer architecture to have a high communication bandwidth between adjacent PEs. The amount of information that is to be transferred between two adjacent processors is proportional to the number of neurons in the layers that are mapped to those processors. The efficiency of implementations utilizing layer level parallelism is related to the match between the neural network structure (number of layers in the network) and the computer system's size and topology. This fact imposes severe restrictions for using this mapping approach for implementing neural networks of varying structures.

2.2.3 Neuron Level Parallelism

By far, most parallel implementations of neural networks have taken advantage of neuron level parallelism. This approach involves mapping each of the neurons in the neural network to a distinct processing element of the architecture, see Figure 2-6. Neurons constitute the fundamental parallel operations in all neural network models. Therefore, the mapping method based on neuron level parallelism can be applied to a wide range of neural network models. This is the major advantage of employing neuron level parallelism over the previously mentioned methods. The main difficulty in achieving practical implementations based on neuron level parallelism is in accommodating the complex and demanding interconnection structures, employed by neural networks, on the limited physical topology of the target computer architecture.

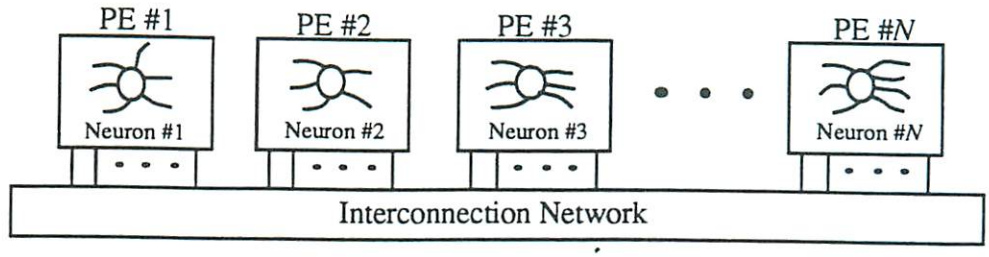


Figure 2-6 - Exploiting neuron level parallelism by mapping individual neurons to distinct PEs.

Due to the large number of neurons in a typical neural network, mapping methods utilizing neuron level parallelism are applicable to medium and fine grain parallel processing architectures. Each processor in the system requires only a small amount of memory since only the synaptic weights associated with a neuron's input or output connections are stored in the local memory of each processor. Since it is impractical to build fine grain systems with fully interconnected topologies, these mapping methods must use multiplexing techniques to map the large number of synaptic connections, employed by neural networks, onto the limited physical connections of the architecture.

The interprocessor communication bandwidth of the systems utilizing neuron level parallelism can be considerably large. Each neuron, mapped to a specific processor, must communicate its output value to other neurons in the network, each mapped to a different processor in the system. The number of data items that are transferred among various processors is thus proportional to the number of synaptic connections used by the neural network. Two primary methods can be used for realizing the synaptic interconnection structures on parallel processing architectures. The first method is based on systolic nearest neighbor communications between processors. In this approach, the regularity of the neural network structures is exploited to efficiently communicate information among neurons mapped to various processors. The second method involves the use of a message-passing network to transfer the neuron output values to their appropriate destination processors. In general, the massage-passing approach requires a longer latency for data transfers between neurons but offers greater flexibility in implementing neural networks with arbitrary interconnection structures.

2.2.4 Synapse Level Parallelism

The finest grain parallel operations that are performed by neural networks are those performed by individual synaptic connections of the model. As mentioned in the introductory chapter, the most common operation performed by a synapse is to multiply its weight value by the associated neuron's output value. By mapping each synaptic connection to a unique PE, all of the operations performed by the synapses can be executed in parallel, see Figure 2-7. As with the neuron level parallelism, synapse level parallelism is applicable to a large variety of models since synapses are also a fundamental part of each neural network model.

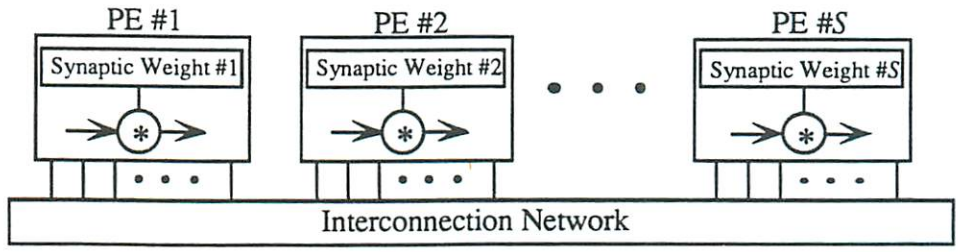


Figure 2-7 - Exploiting synapse level parallelism by mapping single synapses to each PE. In this figure *s* denotes the number of synaptic connections used in the neural network.

In order to efficiently exploit all the available parallelism at the synapse level, the supporting computer architecture should be very fine grain, offering a large number of simple processors. In order to implement the neural computation of equation (2-1), each neuron output value must be broadcasted to all the PEs holding its output synaptic weights. The appropriate weighting function can be applied to the neuron output values by all the processors in unison. Calculating the weighted sum input to a neuron involves communicating and accumulating the values in each processor holding the input synaptic weight of a specific neuron. This scatter-gather type operation is performed in parallel for all the neurons in the network. Such operations require sophisticated routing and communication support. Implementations on architectures that efficiently support synapse level parallelism are rather robust to the neural network interconnection structure since no specific assumption about the network structure is made by the mapping.

2.3 Parallel Implementation Methods

In this section, several methods for implementing neural network models on parallel processing architectures are described. The range of methods used for parallel implementation of neural networks is highly diverse and covers many different levels of design granularity as well as implementation technologies. Among various parallel processing methods used for implementation of neural networks, two principle approaches have been pursued extensively: One in the development of dedicated hardware devices [22, 24, 29, 31, 53, 54, 67, 68], and another in the design of parallel computer architectures and associated mapping methods [5, 9, 11, 12, 18, 20, 25, 30, 34, 35, 41, 43, 50-52, 55, 59-61, 64, 73, 79, 87-89]. The approach to directly mimic the processing of neurons and connections in hardware, either via analog or digital computation, is limited in applicability since these devices offer little flexibility in the type of algorithm and the network interconnectivity structure. These devices have also been difficult to use as building blocks for larger neurocomputer systems since the physical pin-out limitations creates a bottle-neck for transfer of data between various parts of the neural network. Nevertheless, this approach can lead to high processing rates and a relatively compact size suitable for special-purpose implementations. Due to their limited applicability, specific device level implementations are not described in this chapter. However, the various parallel processing approaches and examples described in this chapters can be used as basis or to point out viable directions for further work in physical device implementations.

Among these approaches, implementations on programmable digital parallel processing systems offer the best compromise between flexibility and speed for processing a great variety of neural network models. This approach offers high throughput rates by employing a large number of processing elements. It also offers sufficient programmability for implementing a wide array of neural network models. These systems can be used both for implementing time sensitive processing, and as simulators and development tools for designing new neural network models. A major challenge in the design of such systems is to achieve the maximum amount of flexibility and speed at the minimum cost.

Our presentation in this chapter is mainly focused on methods for implementing neural networks on programmable parallel architectures. In addition to concentrating on

digital parallel processing systems, we further limit the discussion to implementations on Single Instruction stream Multiple Data stream (SIMD) class of parallel processing architectures. The SIMD execution paradigm is an ideal match for implementing neural network models primarily due to the homogeneous nature of the operations being performed by neurons. The SIMD machines are efficient since there is only a single copy of the program being executed by all the processors. The SIMD paradigm allows for simple program development on fine grain parallel machines with minimal overhead in implementing interprocessor synchronization mechanisms. Moreover, limited autonomy in each PE, such as conditional masking and support for a local table lookup mechanism, can significantly increase the efficiency of these architectures for implementing neural networks.

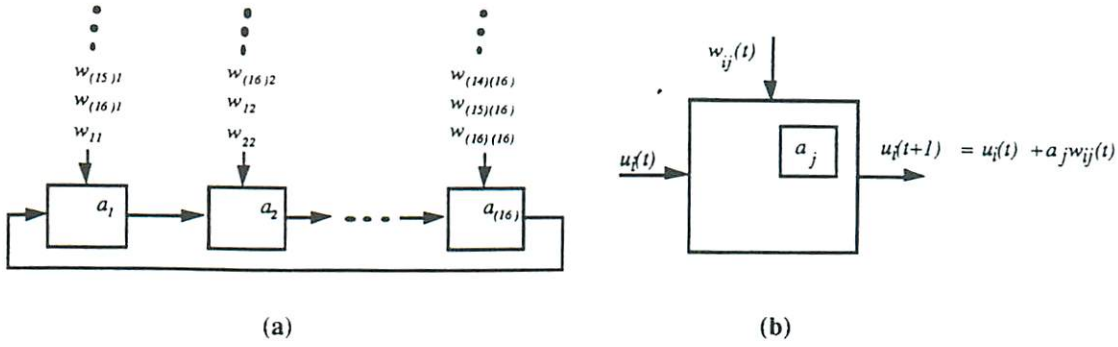
Implementing neural network models on parallel processing architectures requires a method for efficiently mapping neural computations to PEs of the target architecture. In general, the objective of this mapping is to distribute the required computation among the PEs such that the amount of interprocessor communication operations is minimized and at the same time, the number of PEs performing useful operations (i.e. those operations that directly contribute to the neural network processing) is maximized. Many approaches for mapping neural networks onto parallel SIMD architectures have been proposed [5, 9, 11, 23, 25, 42, 55, 60, 61, 64, 73, 88, 89]. The performance of any implementation is strongly related to its efficiency in performing the weighted sum operation of equation (2-2) in a distributed fashion. This requires accumulating the various weighted neuron output values that are distributed across a number of different PEs in the processing array. The implementation of computations requiring data local to the neuron, such as application of the transfer function to the partial sum value, does not require communication among neurons, and therefore, can be processed efficiently in parallel regardless of the interprocessor communication network topology. The presentation below is organized according to the individual mapping method's efficiency in implementing neural networks with specific types of interconnection structures. These include mappings applicable to dense and regularly interconnected networks, sparsely interconnected networks, and any arbitrary interconnected network.

2.3.1 Implementations Applicable to Dense and Regularly Interconnected Neural Networks

Many neural network models are comprised of dense and regularly interconnected neurons, as described in Section 2.1.3. This regularity in the interconnection structure can be used to arrive at efficient implementation methods on regularly structured parallel processing architectures. These methods can be applied to fine grain parallel processors by exploiting neuron level parallelism. The degree of parallelism at this level is equal to the number of neurons in the network (N). Ideally, such a scheme can lead to a similar speedup factor of N. This speedup can only be attained if interprocessor communication delays of the mapping amount to zero. Two distinct methods can be used to achieve zero communication overhead costs. One method is to assume full interconnections among processors of the processing array. Since this approach is prohibitively costly for fine grain architectures, it can be eliminated as a viable option. The second method involves scheduling the flow of data through the processing array such that the appropriate neuron output values arrive at a processor at the correct instant to perform the input sum accumulation operation. This accumulation operation involves summing up a number of neuron output values distributed among the various processors. This type of processing yields zero communication overhead since each processor can perform only a single multiply-accumulate operation during an instruction cycle, and thus, the communication steps can be overlapped entirely by computations.

A very efficient mapping method for implementing neural computation using this approach on systolic array processors has been proposed in [42]. In this method the neurons and the synaptic interconnection network are represented as vectors and matrices, respectively, and the neural computation is formulated as in equation (2-3). This approach treats all neural computations as operations on vectors and matrices. The mapping method takes advantage of the regularities in vector-matrix operations to synchronize the operations of the systolic array. This mapping method exploits neuron level parallelism by assigning each neuron to a unique processor in the processing array. The processors are arranged in a 1-D ring topology with the number of PEs in the machine being equal to the number of neurons in the network. The operations involved in implementing equation (2-3) is depicted in Figure 2-8, where $u_i(t)$ represents the partial weighted sum input to a neuron at time t. The neurons are mapped to processors

and the synaptic weights are organized and accessed in such a fashion that the input value $w_{ij}a_j$ needed by neuron i is added to the partial sum value u_i at the instant that both w_{ij} and u_i arrive at the processor holding a_j.



(a) (b)

Figure 2-8 - Implementing a fully connected neural network with 16 neurons on a ring-connected systolic array. (a) Assignment of neurons to processors and flow of data. (b) Computation performed in each PE at time t.

By utilizing N processors, such a processing scheme can execute N^2 operations (associated with the N^2 synapses) in $O(N)$ time steps. Therefore, properly structured neural networks can achieve optimal system utilization and consequently generate very high throughput rates. Unfortunately, the mapping approach places a number of restrictions on the interconnection network structure in order to achieve maximum efficiency. One such restriction is that the interconnection matrix is required to be quadratic, having equal number of rows and columns, and fully interconnected. Zero weighted synapses and neurons with zeroed output values are introduced to satisfy these constraints, should the network structure be sparse and/or non-quadratic. The efficiency of this implementation method is directly related to the density of the interconnection network since the addition of filler synapses and neurons do not contribute to the actual computation and are used solely to insure proper synchronization of data movement through the array.

This mapping method implements equation (2-3) for a single iteration or a single layer of interconnections. For implementing multilayer neural network, the processing can be performed sequentially using the entire processing array for each of the layers in the network, starting from the input propagating towards the output layer. This approach leads to a processing rate of $O(LN_\ell^*)$ where $N_\ell^* = \max_{1 \le \ell \le L}\{N_\ell\}$, L is the number of layers in the network, and N_ℓ is the number of neurons in each layer. This approach assumes

equal number of neurons per layer and can lead to considerable inefficiencies in implementing neural network models with large discrepancies between the number of neurons in various layers.

Another approach to implementing multilayer neural networks, proposed in [41], involves the use of layer level parallelism in addition to neuron level parallelism. In this approach, each layer is mapped onto a row of a 2-dimensional processing array with wrap-around connections in the East-West directions. Each row of the processing array can then perform the computation associated with each layer and the complete network computation moves in the vertical direction between the rows of the processing array. Using pipelining techniques, L networks can be implemented concurrently to achieve an effective throughput rate of $O(N_t^*)$. This approach can impose even more stringent requirements on the architecture, requiring that the number of rows in the processing array to be equal to the number of layers in the neural network.

2.3.2 Implementations Applicable to Sparsely Interconnected Neural Networks

Although most neural network models are formulated with an assumption of dense interconnection structure, recent findings have demonstrated that performance improvements can be achieved by eliminating a large number of unnecessary synaptic connections [46, 82]. In addition to algorithmic approaches used to prune the neural network structure, a priori knowledge about the specific application can be used to design a neural network with structured but sparse interconnections [70]. Two methods for efficient implementation of sparsely interconnected neural networks, without placing specific demands on the regularity of the network interconnection structure, are described here. The first method, proposed in [61], exploits synapse level parallelism by assigning individual synapses as well as neurons to unique processors of a 2-D mesh-connected parallel processing architecture. The computation method is based on an algorithm for efficient implementation of sparse vector-matrix multiplication operations on 2-dimensional, mesh-connected, SIMD architectures.

The algorithm involves distributing the neuron output values, allocated to various PEs in the machine, to those processors holding their corresponding output synaptic weights. This distributing procedure is performed in parallel via a congestion-free

three-phase routing algorithm using $O(\sqrt{P})$ cycles, where P is the number of processors in the array. When all the neuron output values are received by their associated synapse PEs, a multiplication operation is performed in unison in all the synapse processors taking $O(1)$ time steps. The next step in the processing involves summing the weighted input values of a given neuron, where each value is assigned to a different PE of the machine. The same three-phase congestion-free routing algorithm is used to perform this gathering operation. The execution rate of this mapping method is thus $O(\sqrt{P})$ with a constant multiplicative factor of approximately 24.

Since the number of processors in the array is equal to the number of neurons and synapses of the neural network, the throughput of this mapping method is $O(\sqrt{N+E})$, where E denotes the number of synapses in the network. For a fully interconnected neural network, where $E=N^2$, the asymptotic performance of this mapping is $O(N)$. On the other hand, assuming that in large scale neural network models each neuron is connected to only a constant number (k^*) of other neurons in the network (as is the case with biological neural networks), the number of connections in the network will be $E=k^*N$. Consequently, the throughput rate of this mapping method would be $O(\sqrt{N+k^*N}) = O(\sqrt{N})$. It is apparent that the performance of this implementation method is directly related to the neural network sparsity. Although this implementation method and the one described for ring-systolic architectures [42] in Section 2.3.2 achieve the same asymptotic performance for implementing fully interconnected neural networks, the much smaller multiplicative constant of the ring-systolic method makes it a much more attractive solution for implementation of densely interconnected networks. On the other hand, the three-phase implementation method [61] is more efficient for implementing neural network models with sparse interconnectivity.

Another approach for efficient implementation of neural networks with sparse interconnection structures has been proposed in [84]. This approach exploits neuron level parallelism by assigning every neuron to a unique processor of the system. The computer architecture used for this mapping utilizes the SIMD execution paradigm with nearest neighbor interconnection topology. The fundamental premise of the method is based on having local autonomy such that each processor can independently select one of its neighbors for communication. The architecture is thus similar to that of the DREAM Machine proposed in this dissertation (see Chapter 3). This autonomy in interprocessor

communication is achieved by having dynamically reconfigurable interconnection switches in the nearest neighbor topology.

Having each neuron mapped to different processor in the system, implementing equation (2-2) requires communication among all the processors participating in the evaluation of a single neuron output value. The output value of each neuron is communicated to other processors by a message-passing mechanism using intermediate PEs for routing. In order to assure congestion-free routing, the algorithm creates incrementally longer communication paths at each iteration. This mapping method has been shown to be effective only for extremely sparse networks [12] since the number of useless communication steps grows exponentially with respect to the interconnection density of the network. In addition, with longer paths, the amount of memory required at each PE to support the routing table is also increased exponentially.

2.3.3 Implementations Applicable to Arbitrarily Interconnected Neural Networks

The mapping method described in Section 2.3.1 is efficient for implementing densely connected neural networks while inefficient for implementing sparsely connected networks. The methods described in Section 2.3.2 are efficient at implementing sparsely connected neural networks but not efficient for densely connected networks. In this section, two drastically different mappings are described that are efficient regardless of the interconnection density or structure of the neural network being implemented. The first method, reported in [60], utilizes only network level parallelism to achieve speedup. A complete neural network is mapped to each of the processors of the machine. In the actual implementation on the Warp machine, 10 networks were implemented in each PE to match the computation and communication speeds of the architecture. Weight values were stored in a global memory area and were circularly passed among the various PEs, see Figure 2-9. Since all the values necessary for implementing a given network can be stored locally in each PE, the efficiency of this implementation method is not dependent on the interconnection network structure or density. The throughput rate of this mapping method is thus $O(N^2/P)$. As mentioned previously, network level parallelism is of limited use since it is only applicable for coarse grain architectures where

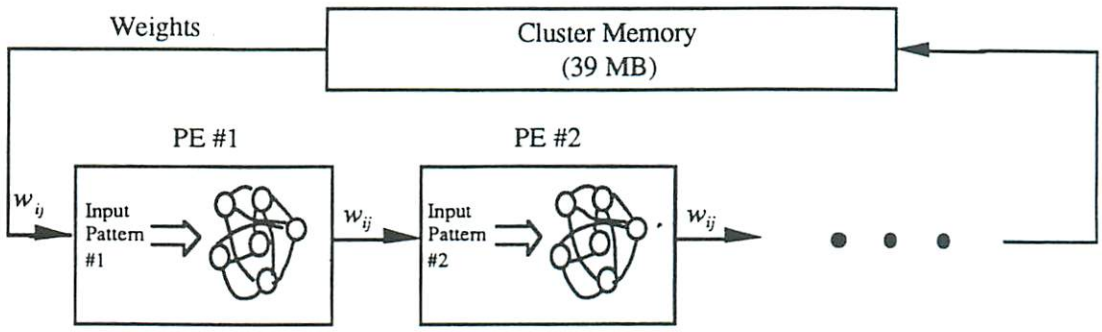


Figure 2-9 - Using network level parallelism for implementing neural networks on the Warp machine.

most often $P<<N$. Consequently, the amount of speedup achieved by this method is negligible for implementing large neural network models.

The second implementation method applicable to arbitrarily connected neural networks has been proposed in [5]. This mapping is in principle very similar to that reported in [61], described in Section 2.3.2. This mapping method has been used for implementing neural networks on the Connection Machine [27], a massively parallel processor. The method exploits neuron and synapse level parallelism by assigning neurons and synapses to distinct processors, see Figure 2-10. Due to the high

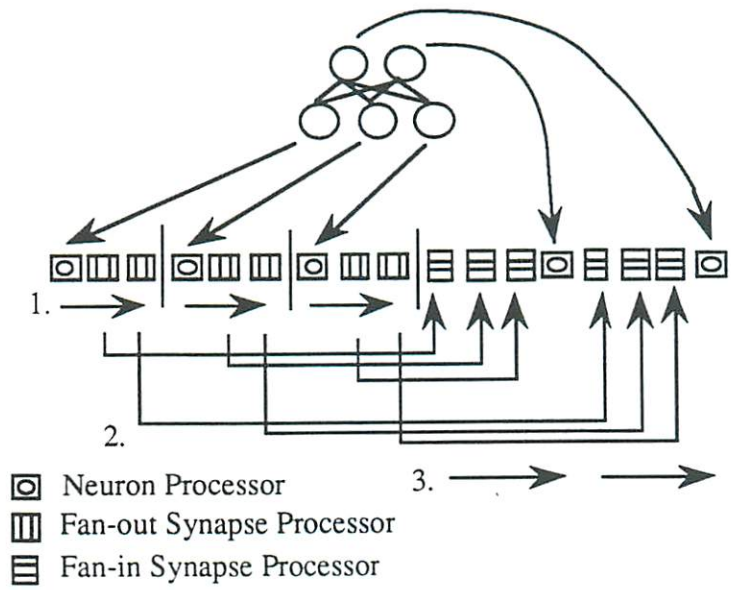


Figure 2-10 - Implementation of neural networks on the connection machine, from [79].

bandwidth interconnection network of the Connection Machine, the scatter and gather operations involved in communicating the neuron output values and summing the weighted input values can be performed efficiently regardless of the number and specific location of the PEs holding the synaptic weight values. Nevertheless, communication overheads associated with this mapping become apparent when the performance of this approach is compared to that of [89] which utilizes both neuron and network level parallelism on the same target machine [79].

Chapter 3

The DREAM Machine

In this chapter, we present the Dynamically Reconfigurable Extended Array Multiprocessor (DREAM) Machine. The DREAM Machine is a parallel processor designed specifically for efficient implementation of neural network models. Due to the considerable diversity in the structure and computational requirements of neural network models and the wide range of applications utilizing these models, the DREAM Machine architecture is designed to be scaleable and to offer programmability in both computation and communication operations. Its computational flexibility is obtained through the use of digital programmable Processing Elements (PEs). The communication flexibility is achieved through the use of programmable interconnection switches used for communicating data between neighboring PEs.

The description of the DREAM Machine architecture presented in the following sections is intended as a general guideline for the design and implementation of the machine. Therefore, specific details of the design, such as clock rates, number of words in the register file, arithmetic function unit design, etc., are left to be determined based on the implementation technology and the allotted cost of the final product. In the following sections we elaborate on special details of the DREAM Machine architecture which are unique to this design and are necessary for efficient implementation of neural network models.

3.1 System Level Overview

The DREAM Machine can roughly be classified as a Single Instruction stream Multiple Data streams (SIMD) [17] medium- or fine-grain parallel computer. The Processing Elements (PEs) are arranged on a 2-D lattice where each PE is connected to eight of its closest neighbors through 4 programmable switches. The top level architecture of the DREAM Machine is depicted in Figure 3-1.

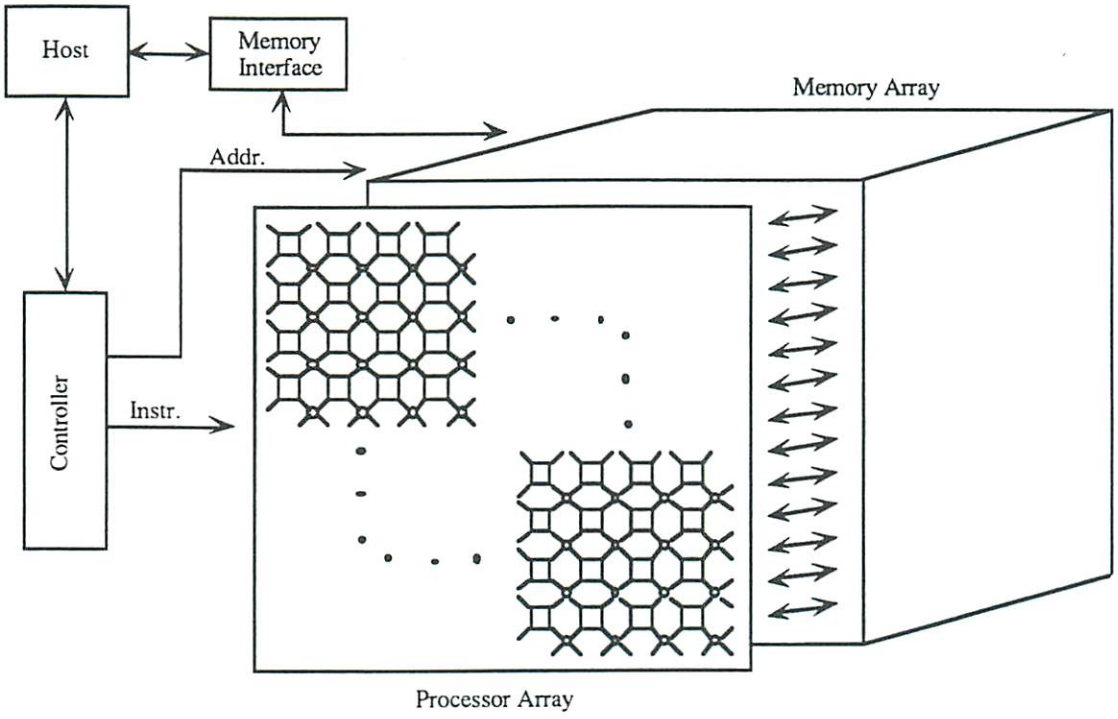


Figure 3-1 - The top level design of the DREAM Machine.

3.1.1 System Organization

The DREAM Machine system consist of three major units: the host computer, the controller, and the Processor Array (PA). The DREAM Machine can be viewed as an attached coprocessor of the host computer. Its memory can be accessed by the host computer through the use of a high speed Direct Memory Access (DMA) channel. The host computer needs only to specify the memory location of the data block to be accessed in the DREAM Machine memory space and the number of words to be transferred. The

DMA controller can transfer the data without using any additional cycles from the host computer or the coprocessor. This feature permits simple programming interface between the host and the DREAM Machine. The host computer is primarily used for properly formatting the input data, long term storage of data, and as a visual interface between the user and the DREAM Machine.

The controller unit interfaces to both the host computer and the PA. The controller contains a microprogram memory area that can be accessed by the host. High-level programs can be written and compiled on the host and the generated control information can be down-loaded from the host to the microprogram memory of the controller. The controller broadcasts an instruction and a memory address to the PA during each processing cycle. The processors in the PA perform operations received from the controller on their local data. Each processor can selectively be masked from performing computation based on a mask flag available in each PE. The DREAM Machine architecture allows for conditionally modifying the status of the mask bit in each processor, giving an additional degree of autonomy to each processor.

The Processor Array (PA) unit contains all the processing elements and the supporting interconnection switches. Each Processing Element (PE) in the PA has direct access to its local column of memory within the DREAM Machine memory space. Due to this distributed memory organization, memory conflicts are eliminated which consequently simplifies both the hardware and the software designs.

3.1.2 Execution Paradigm

The choice of an SIMD execution paradigm for neural computation can be justified by considering the fact that neural network models are built of a homogeneous array of neurons with each neuron performing the same basic operation on a different data set. This type of computation is a perfect match for implementation on SIMD architectures since all processors perform the same computation on their locally stored data. Certain neural network models, such as the Adaptive Resonance Theory (ART) [10] and the neocognitron [19], utilize different neuron types in different layers. In general, the number of different neuron types is small (two in the case of ART and the neocognitron). Furthermore, the majority of operations performed by either type of neurons involve identical processing steps. Therefore, these models can be effectively implemented on a

SIMD machine by time-multiplexing the operations that are specific for each neuron type without a significant loss in throughput. In addition, the use of the SIMD execution paradigm leads to a considerable amount of saving in the instruction memory requirements, since only a single copy of the program is stored for the entire system.

The tightly coupled, distributed memory structure of the DREAM Machine offers a simple scheme for interprocessor communication. Communication between processors is performed one word at a time requiring only a single instruction cycle of each processor. Communication and computation operations can be initiated in parallel by a single instruction. When implementing systolic algorithms, the communication operations can be completely overlapped by computation, and consequently, attain very high execution efficiency.

3.2 Processing Element Design Description

The Processing Element of the DREAM Machine makeup the computational engine of the system. As mentioned earlier, the PEs are part of the Processor Array subsystem. All PEs in the PA receive the same instruction stream but perform the required operations on their own local data stream. Each PE is comprised of a number of Functional Units (FUs), a small register file, interprocessor communication ports, and a mask flag, see Figure 3-2.

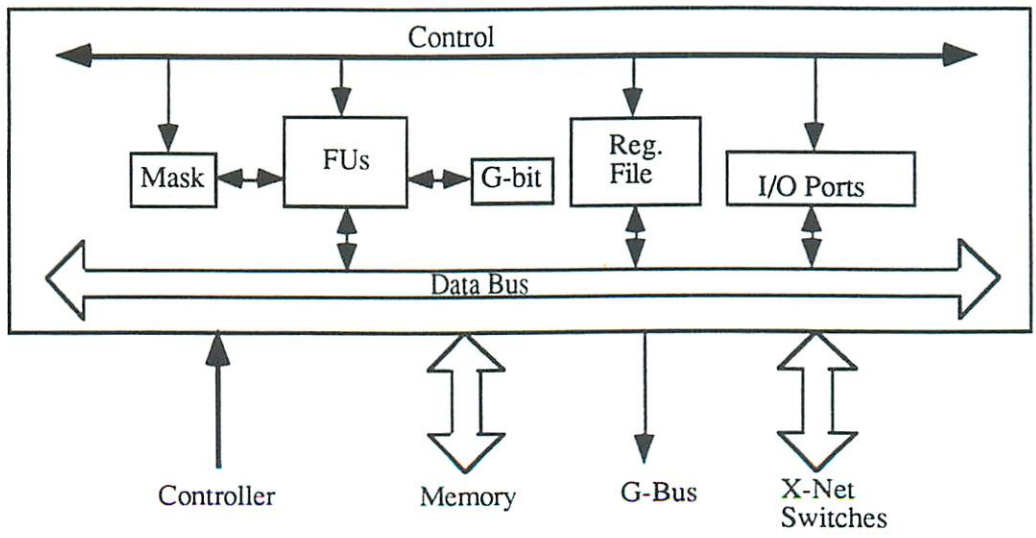


Figure 3-2 - Top level design of the Processing Elements.

3.2.1 Internals of the Processing Elements

The functional units in each PE include adder, multiplier, shift/logic unit. Depending on the specific implementation, additional FUs can be added to the design. Similar to many RISC type processors, each PE has an internal data bus to transfer data between various units. For example, data could move from a register in the register file to one of the operand registers of the adder unit, or data can be transferred from the output register of the multiplier to the I/O output port. A mask bit is used to enable/disable the FUs from performing the operation instructed by the controller. The status of the Mask-bit can be set unconditional by the controller, or determined based on a logical operation performed by one of the FUs. The Mask-bit can be modified during each instruction cycle.

The G-bit register in each PE holds the Most Significant Bit (MSB) of the shift register output. This bit is used to implement two separate functions. The output value of this bit is directly connected to the G-bus, see Section 3.5, and also buffered and used as input to the address modifying shift register in the PE's local memory area. The G-bus connection performs global type operations, such as global broadcasts, finding maximum or minimum activation value, etc.. It is also used to modify the memory address value received by the controller according to a locally stored value. This function is used to implement a variable accuracy lookup table, see Section 3.3.

Each PE communicates with its neighbors through the I/O ports. Only one input and one output port is required in each PE since during each systolic cycle only a single data value is received and transmitted by a PE. The output of each of the I/O registers is connected to the four X-net switches surrounding each PE. The selection of the destination PE for an outgoing data value and the source PE for an incoming data value is made by the specific switch settings of the X-net switches.

3.2.2 Processor-Memory Interface

Each processor of the DREAM Machine has read/write access to its own local memory area, see Figure 3-3. The memory is kept off chip in order to allow for simple memory expansion. During each processing cycle, the memory location associated with each instruction is broadcasted by the controller unit to all the PEs. In this fashion, all the processors can access a single plane of memory at each time step. The memory access

speed is matched with the computation rate of each processor so that each memory access can be completely overlapped with computation. This feature allows for efficient systolic processing.

Each word in the PE's local memory area is comprised of two distinct fields. The first is the data field used to store and retrieve the actual data associated with the computation, such as neuron activation values, synaptic weight values, etc.. The second field holds 4 bits which indicate the switch setting for the X-net switch associated with each PE. Two bits are used for selecting an input port and 2 bits for selecting an output port. These switch setting values are determined prior to the start of computation and are pre-loaded by the host. A new switch setting value can be read during each instruction cycle.

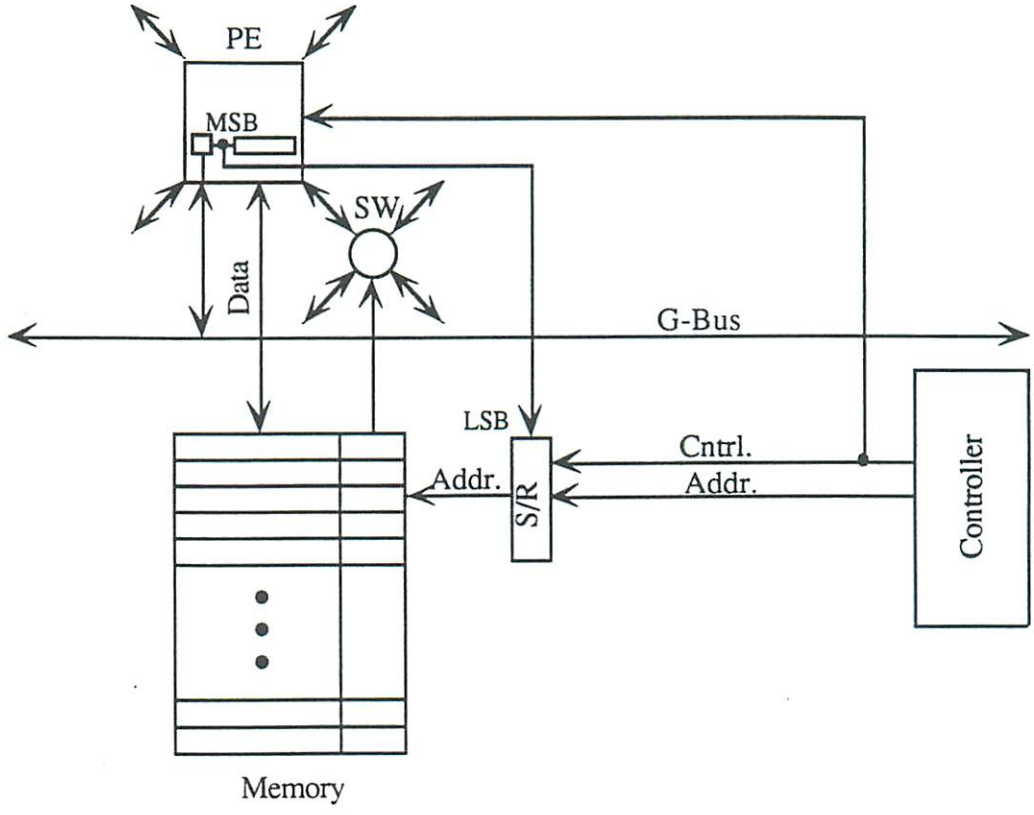


Figure 3-3 - Processor and memory detail diagram. Associated with each data value in the local memory of each PE is a switch setting value used to configure the communication topology of the machine.

3.2.3 Instruction-Word Description

The microword instruction used by the DREAM Machine contains two distinct fields, one for specifying the memory address and another for specifying the operation to be performed by the processors. The instruction set of this machine consists of the conventional arithmetic, logical, and shift operations (e.g. addition, multiplication, AND, OR, left and right shifts, etc.). In addition, the DREAM Machine instruction set also supports global broadcasting of data from the controller to all the PEs in the processing array. This is accomplished using an Instruction/Data bit in the instruction word. If this bit is set to zero, the remaining bits of the instruction word are treated as a single data value. If it is set to one, the remaining bits of the instruction word are decoded and used for their appropriate functions within the PE.

In addition to supplying memory address and control instructions to the PA, each instruction word contains a specific field to control the loading and shifting of data into the memory address modifying register. This field is used when the memory address supplied by the instruction needs to be uniquely modified based on some local information in each processor, as in the case of a table lookup.

3.3 Implementing a Table Lookup Mechanism on the DREAM Machine

A novel feature of the DREAM Machine architecture is its hardware support mechanism for implementing a variable accuracy lookup table. Neural network models use a variety of non-linear transfer functions (represented as f in equation (2-1)), such as the sigmoid, the ramp, and the threshold functions (see Figure 2-2). These functions can be efficiently implemented through the use of a lookup table. Implementation of a table lookup mechanism on a SIMD architecture requires a method for generation/modification of the memory address supplied by the controller, based on some local value in each PE. The BLITZEN Machine [6] performs this task by logically ORing the 10 most significant bits of the memory address supplied by the controller, with a local register value. Such a scheme does not offer sufficient flexibility as required for general-purpose neurocomputer design. The accuracy, or level of quantization of the neuron output values tolerated by

neural networks can vary significantly (from 2- to 16-bits) among different neural network models and different applications of each model.

In order to accommodate lookup tables of varying sizes, the DREAM Machine incorporates two shift registers that are used to modify the address supplied by the controller. One shift register is associated with the PE and keeps the data value used for addressing the lookup table. The other shift register is associated with the PE's local memory and is used to modify the address received from the controller, see Figure 3-3. The table lookup procedure is initiated when the controller loads the base address of the table to each of the shift registers associated with each PE's local memory. The offset value is then shifted into this register one bit at the time from the local register in the PE starting from the most significant bit into the least significant bit of the memory address register. The control signals for this shifting operation are generated by the controller and are broadcasted to all PEs as part of the microinstruction word. With this procedure, an address for a table of size 2^k can be generated in k time steps by each processor.

3.4 Interprocessor Communication Network

Two methods for interprocessor communications are supported by the DREAM Machine architecture. The processors in the PA are arranged on a 2-D lattice with eight nearest-neighbor connections, implemented through dynamically reconfigurable switches. In addition to the nearest neighbor communications performed through these switches, a single bit wide global bus, called the G-bus, is used to perform bit-serial global communications among the PEs.

3.4.1 Nearest Neighbor Communications

Nearest neighbor communications in the PA are performed through dynamically reconfigurable switches connecting one PE to three of its eight nearest neighbors. There are four switches connected to each PE, see Figure 3-4. The basic switch design is similar to the X-net switch used on other systems, such as the MasPar and the BLITZEN machine [6]. The X-net switch allows for communication between eight nearest-neighbor PEs while only requiring 4 I/O connections to each PE. The communication bandwidth

between adjacent PEs are kept equal to the memory access bandwidth to assure efficient systolic processing. The most unique feature of the DREAM Machine architecture is to allow each switch in the interconnection network to be distinctly configured.

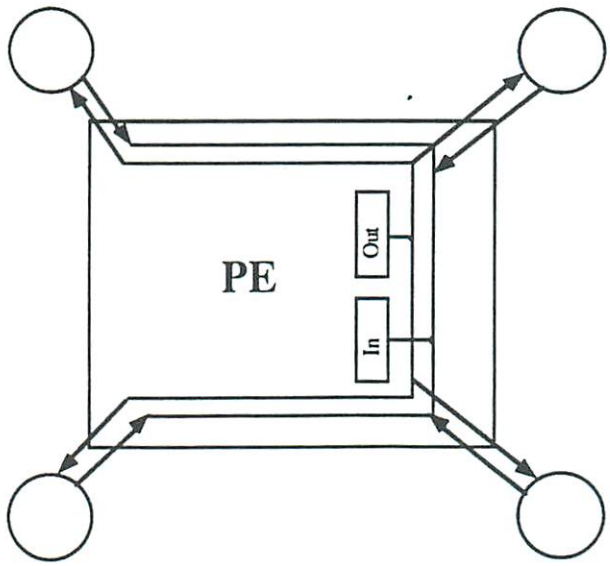


Figure 3-4 - A single processor and its associated interprocessor communication switches.

The switch settings are stored in the local memory of each switch and accessed at the beginning of each processing cycle. The switch memory address is supplied by the controller at each cycle. This design allows for a dynamically changing flow pattern of data through the processing array. In other 2-D connected parallel SIMD architectures, such as the Hughes SCAP [63] and the AMT DAP [57] architectures, all the PEs perform the same communication operation. In other words, the instruction word broadcasted by the controller specifies which direction the data is to be moved in the processing array, e.g. North to South, East to West, West to South, etc.. In the DREAM Machine, on the other hand, one PE can receive data from its North Neighbor while another is receiving data from its West neighbor, depending on the local switch settings. An example implementation of this switch is shown in Figure 3-5 using a multiplexer and demultiplexer to select one of four inputs and route the data to one of four outputs. This flexibility in interprocessor communication is essential in efficiently implementing neural networks with sparse or block structured interconnections, as will be demonstrated in the Chapters 5 and 6.

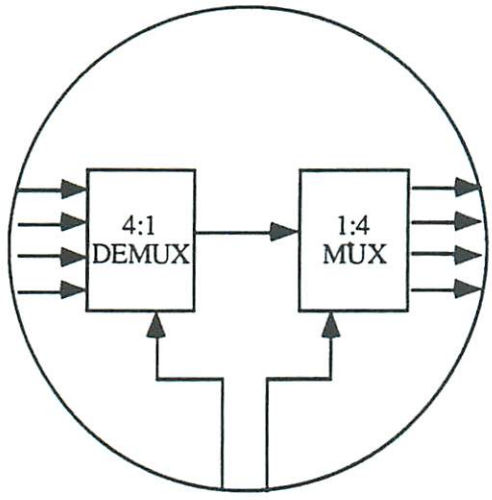


Figure 3-5 - Schematic of the reconfigurable interprocessor communication switch.

3.4.2 Global Communications

Another method of interprocessor communication supported by the DREAM Machine is through a single bit global bus, called the G-bus, electrically connecting all the PEs together, see Figure 3-3. The principal use of the G-bus is to perform global broadcast communications among all processors. The G-bus can be used to efficiently implement competitive neural network models, see Section 5.9. A number of neural network models, such as the self-organizing feature maps [37] and Adaptive Resonance Theory ART [10], utilize various versions of the competitive learning algorithm [39]. Competitive learning requires that the neuron with the maximum activity value be identified. In a distributed system where each neuron is mapped to a different PE, this task might require $O(N)$ time, where N is the number of neurons in the network. Since in general N can be quite large, a more efficient method for determining the neuron with max/min output value is required.

The G-bus is designed as a wired-OR circuit such that each PE can pull down the bus by setting a local flag register to zero. It can thus be used to implement a global minimum or maximum operation in k time steps, where the data to be evaluated is k bits, using the procedure described in [75, 86]. This procedure involves simultaneous broadcasting of the data values over the G-bus by all the PEs one bit at a time starting from the most-

significant-bit. After broadcasting of each bit is completed, all the processors compare their local data value with the wired-OR value on the G-bus. This allows each PE to selectively mask itself out of future broadcasts if another PE in the system has a greater or smaller data value, associated with the min or max function respectively. A useful side-effect of this method of finding the global min/max value is that after the completion of the k processing steps, all the PEs contain the min/max value. In addition to neural network applications, this feature can be useful in implementing other algorithms, such as genetic algorithms [21], where all the PEs require access to the global min/max cost value.

Chapter 4

Mapping Method Preliminaries

In this chapter, we present the basic concepts involved with mapping neural computation on parallel processing architectures. The discussion here is specifically directed at a particular mapping strategy which takes advantage of neuron level parallelism. We state our reasoning behind selecting such a mapping strategy and discuss the various implications involved with its application. The basic principles of the mapping method, assignment of neurons to processors and formation of computational paths, is described in Section 4.1. As a basic starting point, we analyze the performance characteristics of the mapping method used for implementing neural networks on fixed-size ring systolic architectures [42], described previously in section 2.3.1. This analysis demonstrates that the mapping method can be optimum, in the sense of full utilization of system resources, in some specific applications. At the end of this chapter, we present an extension to this basic mapping method, utilizing network level as well as neuron level parallelism, in order to enhance the performance of the mapping when applied to 2-D connected parallel processing architectures. In Chapter 5 and 6, we present two mapping methods which are based on a generalized version of the basic mapping described in this chapter. These new mapping methods can be applied with much greater efficiency to implement a larger number of neural network interconnection topologies.

4.1 Mapping Principles

The mapping method described in Section 2.3.1, for implementing neural networks on ring connected systolic architectures, can be very efficient in principle. Given full

interconnections between all of the layers in the neural network and a perfect match between the processor ring size and the neural network size, this mapping can achieve 100% system utilization and therefore a linear speedup factor. This type of utilization is possible since the mapping has the potential to keep all of the processors in the array busy performing useful operations. Inefficiency in the mapping arises in the cases where the network structure is not regular and its size does not math the processing array size. In such cases NOP operations are used to synchronize the flow of data to conform the neural network with the processor array. In order to reduce the effects of the neural network structure on the implementation efficiency, we now propose a generalized version of this mapping method. Due to the use of neuron level parallelism, this mapping method can be applied efficiently to a great variety of neural network models.

4.1.1 The General Mapping Method

In our general mapping approach, individual neurons are assigned to distinct processors of the parallel architecture. For the sake of simplicity, at this time we assume that the number of processors in the target machine is always greater than or equal to the number of neurons in the network. The cases for which this assumption is not valid are discussed separately for each mapping method in Chapters 5 and 6. In general, the assignment of neurons to processors can be performed arbitrarily. Performing the neural computation operation of equation (2-1) involves the calculation of a weighted sum of all the inputs to a specific neuron. Since all of these inputs are output values of other neurons in the neural network, each value to be added is located at a different processor in the machine. Thus, the summation operation can be performed in a distributed fashion.

A small segment of an arbitrarily connected neural network and an example mapping is shown in Figure 4-1. The basic computation performed by each processor in the machine is identical to the one described in Section 2.3.1. More specifically, a PE holding the output value of neuron j receives the partial sum value $u_i(t)$ from the previous PE in the path at time t. It then adds its contribution $(w_{ij}a_j)$ to the partial sum value $u_i(t)$ and passes on the updated partial sum value $u_i(t+1)$ to the next PE in the path. Due to the commutative nature of the addition operation, the computation paths required for evaluation of the partial sum values u of equation (2-2) can be constructed without any specific ordering of the processor traversals, as long as each path passes through all the

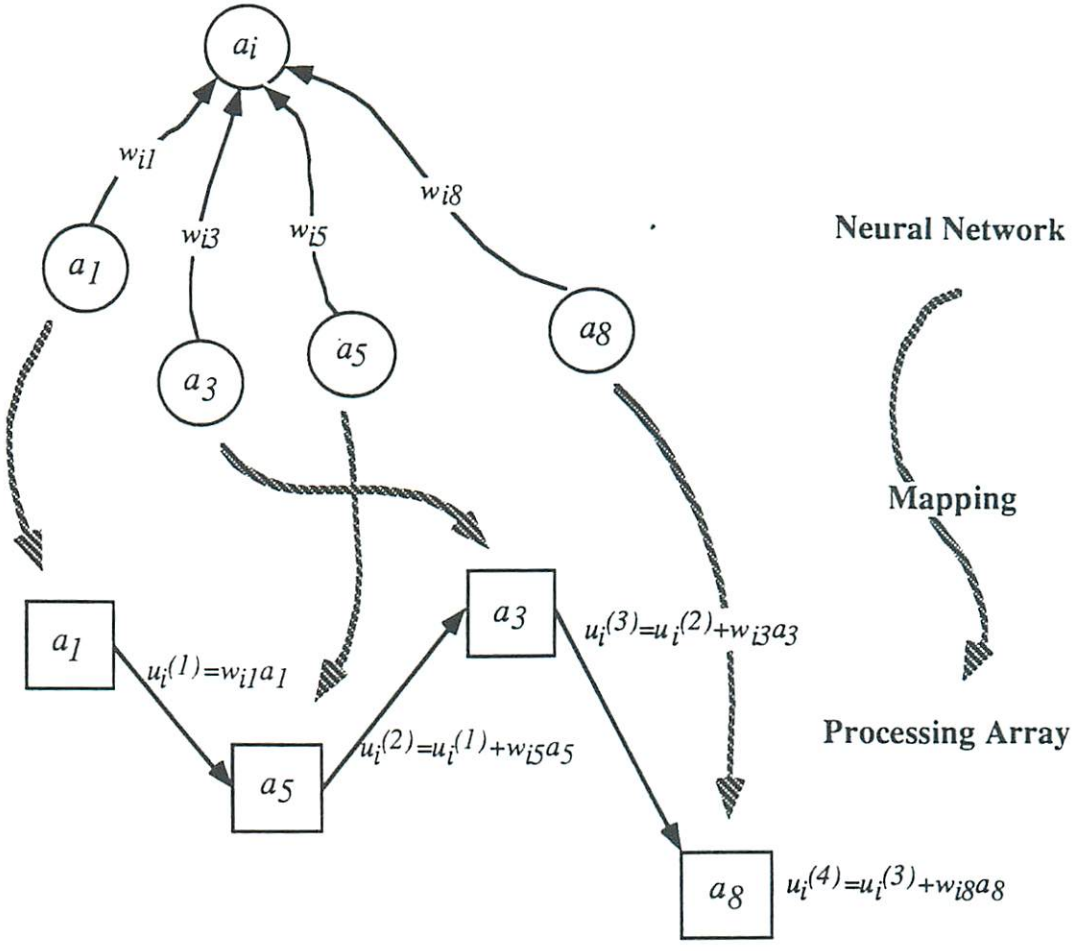


Figure 4-1 - Mapping neurons to PEs and constructing a computation path between the PEs.

PEs contributing to its partial sum value u_i. Moreover, multiple paths can be traversed concurrently as long as no two paths attempt to cross the same PE at the same time.

4.1.2 The Algorithmic and Optimized Mapping Approaches

The mapping method described in the previous section describes how the weighted sum operation of equation (2-2) can be implemented in a distributed fashion. However, this general mapping does not explicitly specify the exact assignment of neurons to processors and the formation of computational paths through the processing array. The simplest method is to linearly assign the neurons to processors, as described in [42]. If the

processors are arranged in a linear ring topology, creation of computational paths is simply accomplished by having each path start at a different PE and travel in the same direction around the ring. This will guarantee conflict-free communications among the processors. Unfortunately, this method, as mentioned previously, can be very inefficient in implementing irregularly connected and sparse neural network structures.

In this dissertation, we propose two methods for efficiently arriving at assignments of neurons to processors and creation of conflict-free computational paths. The first method is called the algorithmic mapping method. This method is similar in spirit to the linear ring mapping, except it allows for construction of multiple processing rings of variable size on the 2-D connected processing array of the DREAM Machine. This method is suitable for mapping neural networks with regular interconnection structures. It can also address sparsely connected neural networks efficiently as long as the interconnection structure is comprised of a number of densely connected blocks of neurons. The second mapping method, on the other hand, involves a combinatorial optimization procedure. Since the throughput of a mapping can be measured with respect to the specific assignment of neurons and computational paths, optimization methods can be used to find optimal or close to optimal mapping solutions. Therefore, a method for analytically measuring the throughput rate of a specific mapping is required. In Chapter 6, we present one such measure in the form of an energy cost function. A neural-type optimization method, called Constrained nets, is used to find solution mappings that are close to optimal or at times optimal.

4.2 Mapping Neural Networks onto Fixed-Size Ring-Connected Architectures

A simple yet efficient mapping strategy for implementing densely interconnected blocks of neurons can be devised by linearly assigning each neuron to a single PE of a ring-connected processing array, as shown in Figure 4-2. In order to simplify the discussion, we describe the mapping for a single block of a two layer network consisting of only an input and an output layer. The extension of this method to multilayer networks will be addressed later in section 4.2.3. The basic mapping concept is to assign individual neurons and all of the weights associated with their output synaptic connections to unique processors. The mapping method requires that the PE being

assigned the first neuron of the block to be a neighbor of the PE that is assigned the last neuron of the block. This restriction implies construction of a processing ring of size equal to the number of input layer neurons. In cases where the number of output layer neurons is greater than the number of neurons in the input layer, the input layer is padded with neurons having zero output values so that the input and output layers are of equal size.

For the sake of simplicity, we will assume that the number of processors in the system is greater than or equal to the number of neurons. In order to evaluate the output

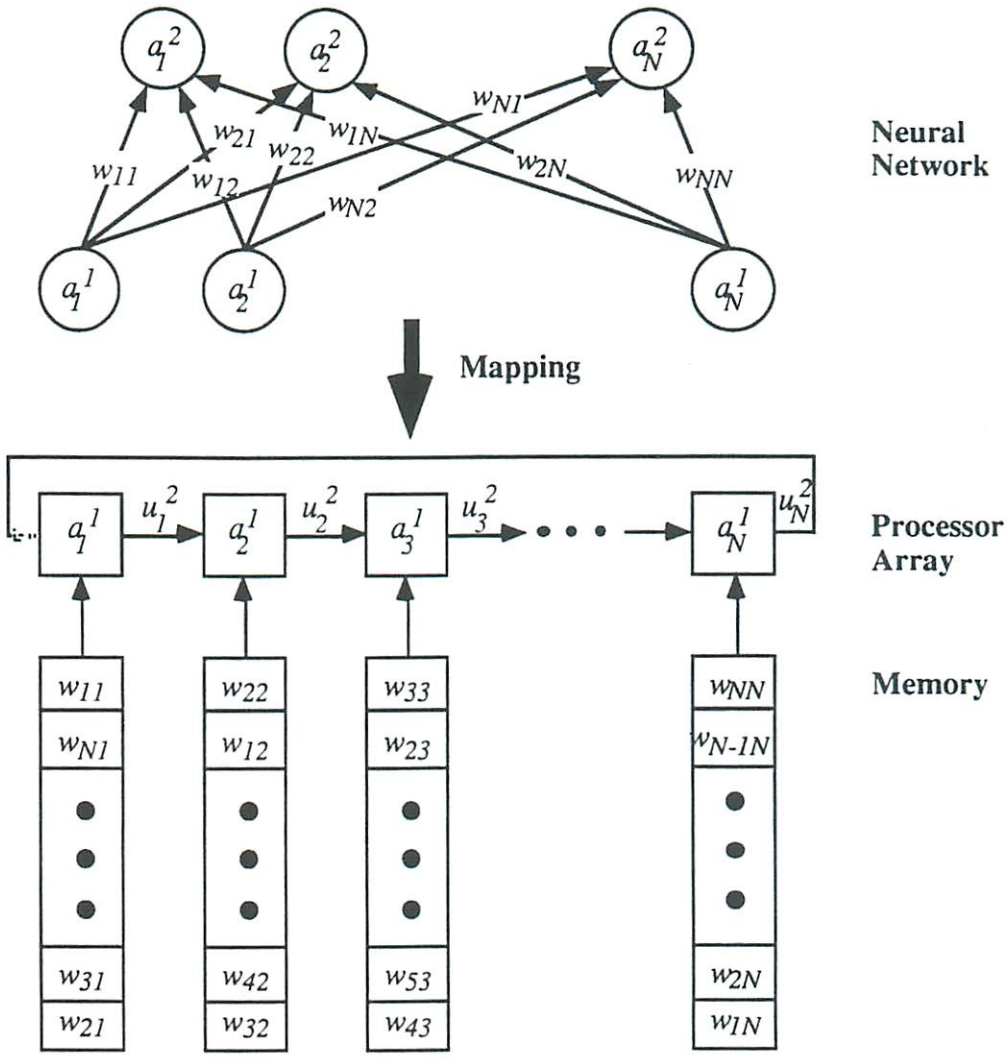


Figure 4-2 - Mapping a fully connected neural network onto a ring systolic processing array. Note: Superscripts denote layer numbers.

value of each neuron in the output layer, according to equation (2-1), the weighted sum of all the input layer neuron's output values must be calculated. As mentioned in Section 2.1.1, this is a distributed operation requiring data from each of the neurons assigned to different processors. Due to the inherent serial nature of the addition operation, for each output layer neuron, we can construct a path through all the PEs participating in this summation operation by starting at a specific PE and traversing the complete ring arriving back at the same starting PE. Each path is initiated in a different processor and rotates in the same direction around the ring, thus always insuring that no two paths ever cross the same PE at the same time.

At each time step t, the product $w_{ij}a_j$ is calculated by the i^{th} PE in the ring and added to the partial sum value u_i received from the preceding PE. The partial sum is accordingly modified as $u_i(t+1) = u_i(t) + w_{ij}a_j$ and passed to the next PE. After a complete traversal of the processing ring, the appropriate threshold value θ_i can be added to the partial sum u_i followed by the application of the transfer function f. In this mapping, only a single systolic cycle of each processor is needed by each path to perform the multiply-accumulate operation, therefore, multiple paths can utilize the same PE each at a different time. This feature allows for parallel execution of various paths in the network. This mapping is equivalent to the one described in Section 2.3.1. Similar mapping approaches have been proposed in the literature for implementing neural network models on various parallel processors [55, 73, 89].

4.2.1 System Utilization Characteristic of the Mapping Onto the Fixed-Size Ring Architecture

Mapping neural networks onto ring-connected processing arrays is highly efficient when the number of neurons in the network is equal to the number of processors in the processing array. Formally, let N_ℓ represent the number of neurons in layer ℓ, and P represent the number of PEs in the processing array. Since P paths are traversed simultaneously with each path performing P operations, the implementation has a potential for performing P^2 useful calculations. A two layer neural network requires $\mu N_\ell N_{\ell-1}$ operations where μ is defined as the interconnection density between layers ℓ and ℓ-1, calculated by

$$\mu = \frac{\#\text{ of non - zero connections between layer } \ell \text{ \& layer } \ell - 1}{N_\ell N_{\ell-1}}. \qquad (4\text{-}1)$$

Assuming that the number of neurons in each layer of the neural network is less than or equal to the number of PEs in the processing ring, the system utilization factor η indicating the ratio of useful operations per total possible operations can be determined by

$$\eta = \frac{\mu N_\ell N_{\ell-1}}{P^2}, \qquad (4\text{-}2)$$

where $\mu N_\ell N_{\ell-1}$ is the number of non-zero synaptic connection weights in the neural network. It is apparent that the maximum system utilization ($\eta = 1$) is achieved when μ=100% and $N_\ell = N_{\ell-1} = P$.

System utilization η scales linearly with respect to both interconnection density and the ratio of neurons to PEs. Should the number of neurons exceed the number of processors, virtual processors can be created using time multiplexing techniques. In such cases a single PE can simulate multiple processors by using different consecutive time slices to implement the processing associated with each virtual PE. The general form of the system utilization function for a ring systolic array can thus be written as

$$\eta = \frac{\mu N_\ell N_{\ell-1}}{\left\lceil \frac{N_\ell}{P} \right\rceil \left\lceil \frac{N_{\ell-1}}{P} \right\rceil P^2}. \qquad (4\text{-}3)$$

where $\lceil N_\ell / P \rceil$ indicates the number of neurons of layer ℓ assigned to one physical processor.

It can be noticed that the round-off factor introduced by the ceiling function can be minimized by either keeping the number of neurons per layer N_ℓ equal to an integer multiple of the processing array size P, or by having much fewer processors than the number of neurons in the network. With the round-off effects removed under these conditions, the system utilization will be

$$\eta \approx \mu. \qquad (4\text{-}4)$$

This result indicates that to insure good system utilization, regardless of the exact size of the neural network, we require dense interconnections between layers of neurons and a small number of processors. At the same time, we would like to exploit all the available parallelism in the neural network by having a large number of processors. This causes a dilemma in determining a good number of PEs to be used in the system architecture.

Large number of processors is required to effectively implement large neural networks, but their use causes great inefficiency for implementing networks with different sizes.

4.1.2 Execution Rate Characteristics of the Mapping Onto the Fixed-Size Ring Architecture

Although system utilization is of interest in evaluating the efficiency of a particular implementation, the primary measure of performance is the execution rate of the implementation. We will now analyze the execution rate of the mapping and the effects of processor array size on system throughput. The number of cycles required to process a neural network using the above mentioned mapping on a ring systolic architecture is equal to the number of cycles needed to traverse a path (making a complete circle around the processing ring) plus the additional cycles used in implementing the neuron activation function. The total time for implementing a single block can be calculated by

$$T_\ell = \left(\left\lceil \frac{N_\ell}{P} \right\rceil \left\lceil \frac{N_{\ell-1}}{P} \right\rceil P k_1 + \left\lceil \frac{N_\ell}{P} \right\rceil k_2 \right), \tag{4-5}$$

where k_1 is the time required to perform a multiply-accumulate operation and k_2 is the time required to implement the neuron transfer function. This formula also assumes the use of time-multiplexing of multiple neurons on a single PE in cases where the processing array is smaller than the neural network. If $P \geq N_\ell$ and $P \geq N_{\ell-1}$ the execution time will be $O(P)$.

This implies that it is possible to reduce the execution rate of an implementation by increasing the number of PEs in the system. As the number of neurons in the network increases, we would like to increase the number of PEs to maintain high throughput. Unfortunately, with a large number of processors, the degradational effects of unequal number of neurons to PEs on the system utilization and throughput is proportionally larger. In Chapter 5, we will outline a method for constructing variable-length rings on the DREAM Machine which can alleviate a number of these problems by matching the ring size to the problem size.

4.2.3 Mapping Multilayer Neural Networks Onto the Fixed-Size Ring Architecture

Implementing multilayer neural networks on 1-D ring systolic arrays can also be performed using the mapping technique described above through time multiplexing of the processing associated with each layer on the processor array. The computation associated with each layer of a neural network can be performed consecutively on the ring array as the data flows from the input layer toward the output layer. This type of implementation leads to an execution time of $O(LP)$, where L is the number of layers in the neural network structure. The exact execution time can be calculated as

$$T = \sum_{\ell=2}^{L} T_\ell , \tag{4-6}$$

where T_ℓ is defined by equation (4-5).

Pipelining techniques can be used, as proposed in [Kung, 1989 #14], to decrease the execution time back to $O(P)$ given L ring arrays organized in a 2-dimensional mesh topology. The efficiency of this type of implementation is strongly related to the match between the number of neurons and layers to the size and structure of the processor array. The pipelined mapping method requires the number of layers in the neural network to be equal to the number of rows (i.e. pipeline stages) in the processor array. Furthermore, the mapping assumes equal number of neurons for all the different layers. These are rather strong restrictions on the neural network interconnection structure for general-purpose implementations.

4.2.4 Deficiencies of the Mapping Onto the Fixed-Size Ring Architecture

As mentioned previously, a major deficiency in the use of the linear mapping method for implementing neural networks on fixed size ring systolic architectures is attributed to the strong dependence of the processor array size on the efficiency in implementing varied size neural networks. We have shown through equations (4-5) and (4-6) that the execution rate of this mapping is $O(P)$ when both the processing array size and neural network size are of the same order of magnitude. This type of scaling characteristics is not well suited for neural network applications since the large number of neurons required

for real-world applications will in turn require large number of processors which would finally yield a large execution rate.

If we assume full interconnections between all the neurons of a large real-world network, this mapping would remain viable as it yields maximum utilization. However, current evidence, both from biological examples [3] and artificial neural networks [46], indicate that real-world systems are constructed of modular and sparsely interconnected networks. This fact thus indicates that the fix-ring mapping method, although useful for relatively small and densely interconnected neural networks, is not suitable as an implementation method for large scale general-purpose neural processing.

In addition to the low system utilization characteristics of the fixed-size ring systolic mapping, the processing array interconnection topology is also problematic. The ring systolic architecture utilizes a 1-D interconnection topology which does not lend itself well to such applications as image processing and vision where neural networks have extensively been used [19, 83]. A planner topology is a more natural architecture for such applications where the data is structured and manipulated in a 2-D format. In addition, locally connected 2-D planar architectures, such as the mesh, are a more natural match to VLSI implementations than the 1-D topology due to the inherent 2 dimensional nature of integrated circuits.

4.3 Initial Mapping Method for 2-D Connected Architectures

A mapping method suitable for 2-D VLSI implementations based on the fixed-size ring method has been demonstrated in [72]. The basic motivation for this mapping method was initially to extend the use of the ring systolic mapping to be applicable mesh connected processing arrays. In addition, we required the mapping method to have small local memory requirements in order for it to be used with fine-grain parallel architectures. The basic concept of this mapping involves the use of batch-mode processing, where different instantiations of a complete network are mapped onto different rows of a 2-D processor array, see Figure 4-3. Each row of the array is considered as a single 1-D ring of processors employing a mapping similar to the one described above in Section 4.2. The computation performed in each row is similar to that of the mapping in [42] except

that the weight values are communicated between various networks through the vertical connections of the processing array.

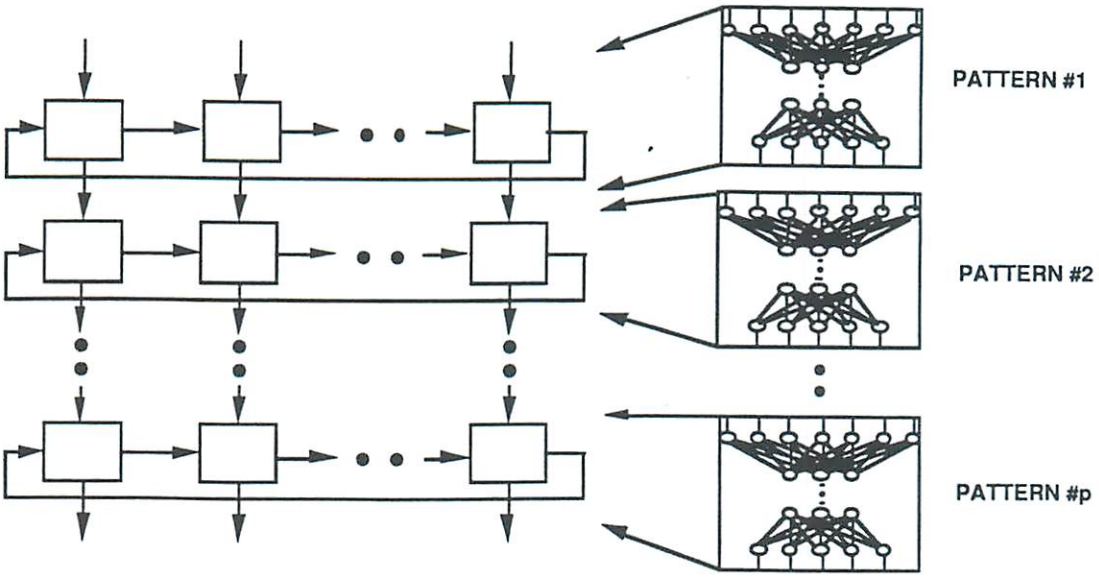


Figure 4-3 - Exploiting network level and neuron level parallelism on a 2-D mesh connected systolic architecture.

This mapping is memory efficient since only a single copy of the interconnection weights are stored and used by all the networks in the various rows of the array. Computation associated with each layer of the neural network is performed sequentially in each row. This imposes no constraints on the number of layers in the neural network, but the mapping nevertheless requires equal number of neurons per layer and full interconnections between neurons of adjacent layers. Very high execution rates have been reported using this type of mapping approach for implementing densely connected neural networks [73, 89].

In comparison to the fixed-size 1-D ring systolic mapping described in Section 4.2, our mapping method here is suitable for a large class of neural networks and fine- to medium-grain, two-dimensional, mesh-connected, SIMD arrays. However, it is described in detail here for implementing the multilayer perceptron neural network using the back-propagation learning model. The performance of the method is illustrated using the Nettalk neural network and the Systolic/Cellular coprocessor of Hughes [62] in order to allow for comparison with other implementations.

In this section we describe the implementation of the data processing in the recall and learning phases of the multilayer perceptron (MLP) network with back-propagation learning (also called the back-prop model) on a 2-D mesh array architecture. The structure of the MLP network consists of several layers of neurons, where the first layer is called the input layer, the last is called the output layer, and all the layers between the input and output layers are called hidden layers. There are no synaptic connections between neurons on the same layer, but each neuron is assumed to be connected to all other neurons in the adjacent layers.

4.3.1 Computational Aspects of the Back-Propagation Algorithm

The back-prop model operates in two distinct phases, one is the recall phase in which the training pattern is presented to the input layer of the network and a corresponding output is *recalled* at the output layer. The other is the learning phase in which the network adjusts its synaptic weights in order to minimize the error between the recalled pattern and the correct pattern supplied by a supervisor. The operations of each neuron in the network during the recall phase is demonstrated in Figure 4-4, where a denotes the

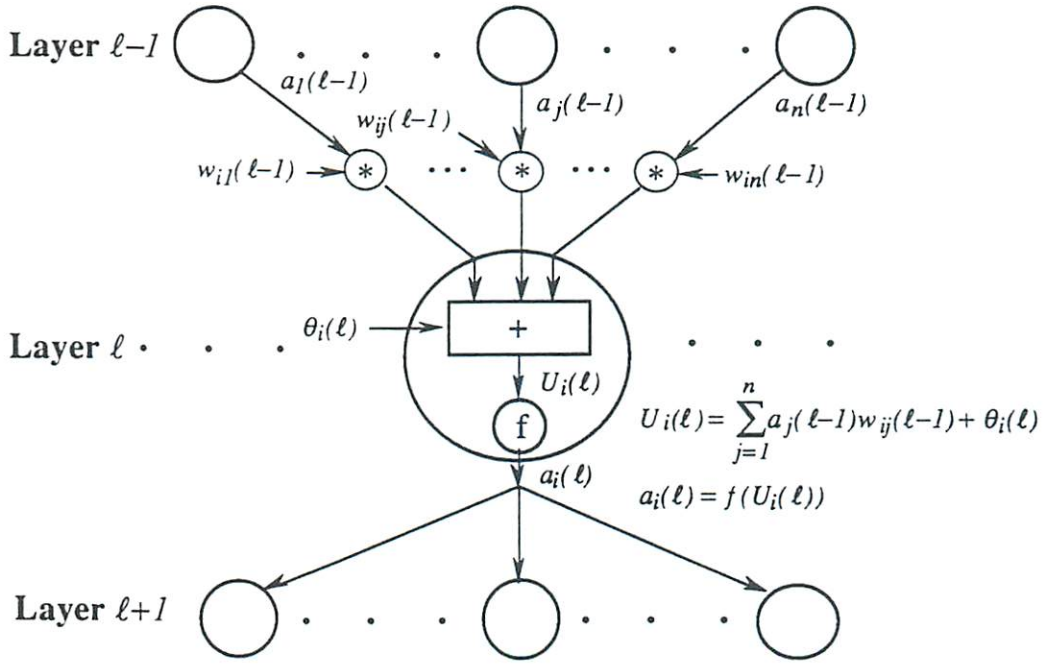


Figure 4-4 - Recall phase (forward pass) processing of neuron i on layer ℓ of the multilayer perceptron neural network.

neuron activation value, w denotes the synaptic weight value, θ denotes the neuron threshold value, and f denotes the nonlinear neuron activation function. In short, the function of each neuron, during the recall phase, is to calculate the weighted sum of all its inputs and apply the nonlinear activation function to this sum in order to generate the neuron's output value.

Similarly, the processing involved in the learning phase is depicted in Figure 4-5, where δ denotes the error term and f' is the first derivative of the neuron activation function. The error terms for neurons on the output layer (ℓ=L) are calculated as

$$\delta_i(L) = f'(U_i(L))(t_i - a_i(L)) \qquad (4\text{-}7)$$

where t_i denotes the desired output value of neuron i supplied by the supervisor. Whereas in the recall phase, the neuron activation values are propagated forward in the network, the error values are propagated backwards in the network during the learning phase. During the backward propagation of errors, the synaptic weight values are modified according to

$$\Delta w_{ij}(\ell) = \eta \delta_i(\ell+1) a_i, \qquad \text{where} \qquad (4\text{-}8)$$

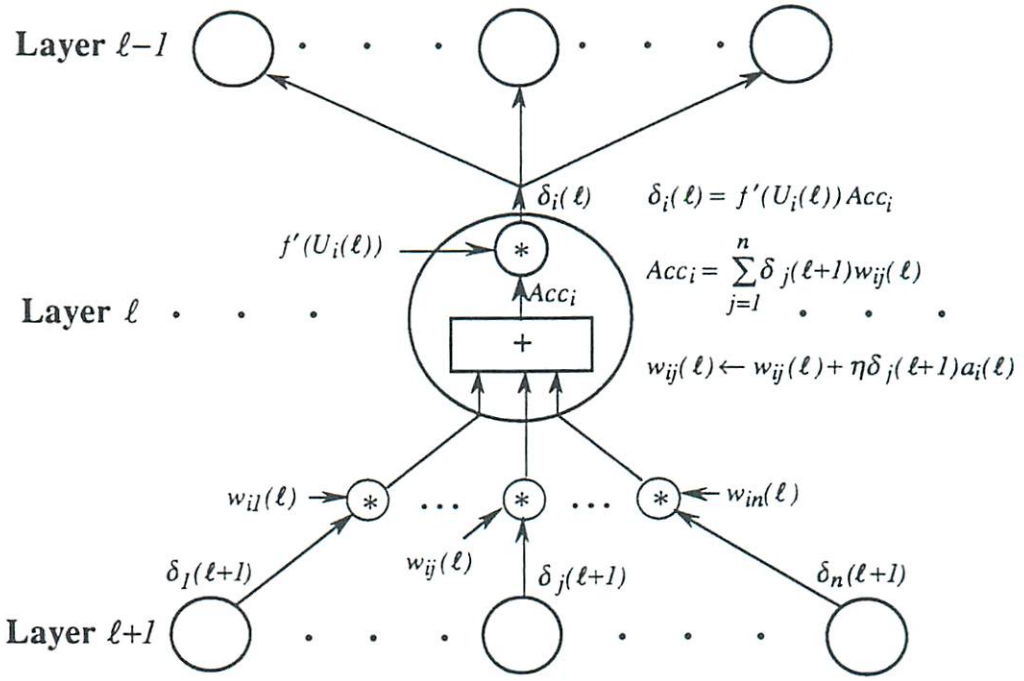


Figure 4-5 - Learning phase (error back-propagation) processing for neuron i on layer ℓ of the back-propagation algorithm.

$$\delta_i(\ell) = f'(U_i(\ell)) \sum_j \delta_j(\ell+1) w_{ji}(\ell) \qquad \text{for } \ell < L. \qquad (4\text{-}9)$$

In the remainder of this section we introduce a mapping method for efficient implementation of the operations in the recall and learning phases of the back-prop model on a 2-D mesh-connected SIMD architecture. For the sake of simplicity, let us assume for now that the neural network under consideration has the same number of neurons in each layer, and that this number is equal to the number of processors in a single row of the processor array. This assumption will be dropped later.

4.3.2 Mapping Details

In our method, a different input pattern for the entire neural network is mapped onto each row of the processor array, as shown in Figure 4-3. Thus, each row processes data for one complete network starting from the first layer and then continuing sequentially for consecutive layers until the outputs are obtained. At the beginning of the recall phase, the processors contain, in their local memory, the inputs for the network, one input

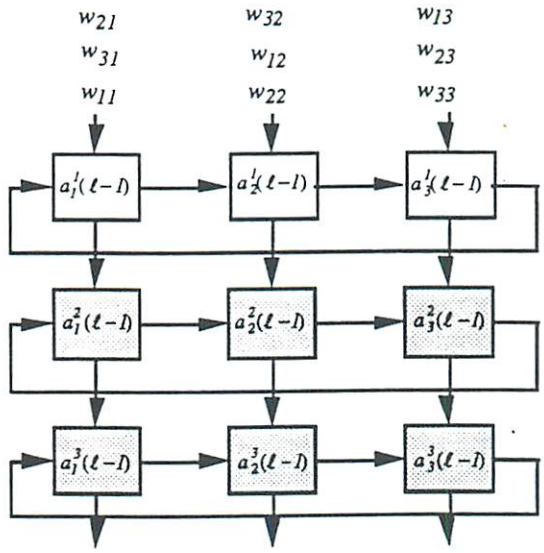


Figure 4-6 - Data organization in the processor array during the first cycle of the recall phase. Only processors in the first row are enabled.

pattern per row and one item per processor, see Figure 4-6. The indices of the three different sets of inputs/activation values appear as superscripts, subscripts denote

different neurons in a given layer, and the layer index is given in parentheses. The shaded squares represent the processors that are disabled (masked) in the given cycle.

The different input patterns are transformed into outputs of the first hidden layer of the networks by the processing involving vertical - North to South - flow of the synaptic weights w_{ij}, and horizontal flow - West to East- of the partial sums U_i . A single shift South is followed by a single shift West until the partial sums make a full circle. One processing cycle, for the j^{th} processor, consist of the following operations: (1) receiving the weight value w_{ij} from the North neighbor and at the same time, sending the weight value w_{ij} received in the previous cycle to the South neighbor; (2) receiving the partial sum U_i from the West neighbor and at the same time, sending the old U_i to the East neighbor; (3) Multiplying the incoming w_{ij} by the a_j, which is stored locally, and adding it to the incoming U_i, see Figure 4-7.

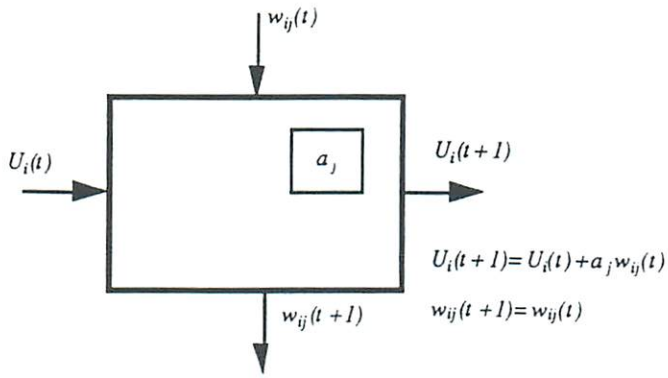
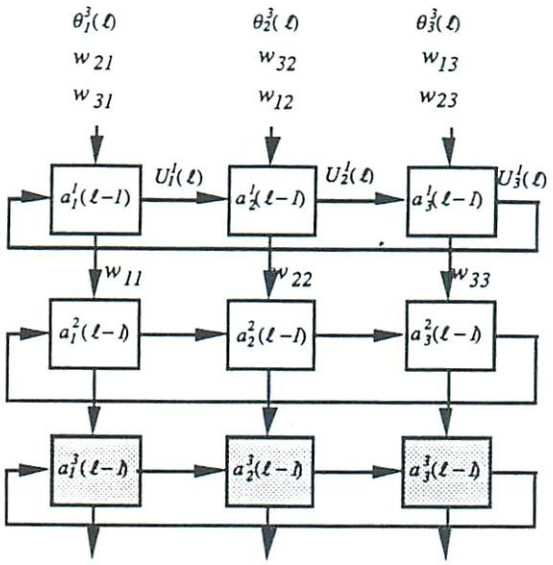
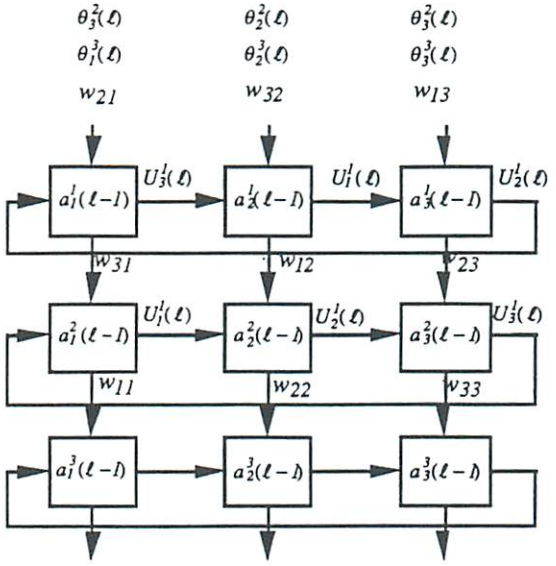


Figure 4-7 - Processing in each processor of the array during the recall phase.

The weight values are organized in a skewed fashion in the global memory. This organization is used in order to have the correct weight value w_{ij} and partial sum U_i meet in the same processor at the proper time. Figure 4-8 demonstrates the next two iterations following the initial cycle shown in Figure 4-6.



(a)



(b)

Figure 4-8 - Next two cycles of the data flow through the array after initial cycle shown in Figure 4-6.

In this procedure, the vertical flow brings the same weights in contact with the different input patterns in the consecutive rows of the array. This is possible since the weight values are identical for all the networks in the array. Thus two levels of parallelism are achieved, one is the processing of complete networks being performed

concurrently in a pipeline mode in the different rows of the processor array, and two is that each network is being processed in parallel by the different processors of each row. After all the weight values of a given layer have passed through the processor array and the threshold values θ_i have been added to the partial sums U_i in each row, the computation of the activation function takes place in unison in all the processors. At the completion of this operation, the activation values $a(\ell)$'s for a specific layer ℓ are available in each processor for computing the activation values for the next higher layer $\ell+1$. Concurrent with the propagation of weights through the array, the final U_i values in each row are propagated through the array and stored in the global memory for later use during the learning phase. This process is repeated until the activation values for the neurons on the output layer have been evaluated.

It is apparent from Figures 4-6 and 4-8 that the processing of multiple networks in the processor array is performed in a pipeline fashion. The organization of data in the memory is structured such that the pipeline flushing time is overlapped with loading of values for the next operation. This method is efficiently used to implement networks larger than the processor array by loading and unloading different segments of the network for processing.

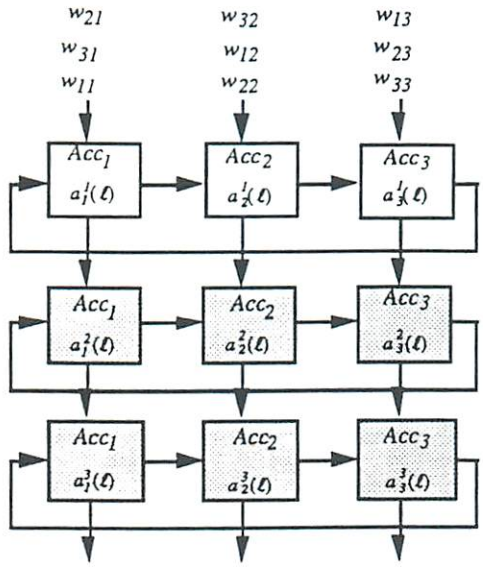


Figure 4-9 - Data organization in the processor array during the first cycle of the learning phase. Only processors in the first row are enabled.

During the learning phase, we also assign the computation of an entire network to each row of the processor array. This leads to a similar data flow and identical

organization of the activation and weight values. Thus the transition from the recall to the learning phase does not require data reorganization. The activation values a_i along with the accumulated sums Acc_i and the learning rate η are stored in each processor's local memory, see Figure 4-9.

The weights w_{ij} travel from North to South, as before, and the error values δ_i move from West to East, see Figure 4-9. Two sets of weights are being transferred concurrently - the old and the new, where the new weights ω_{ij} are computed on the fly according to the learning (or weight update) formula from Figure 4-10. The old weights w_{ij} are the same weight values as used during the recall phase. Treating the new and the old weights separately in this fashion insures consistency in learning between multiple networks executing in different rows of the processing array.

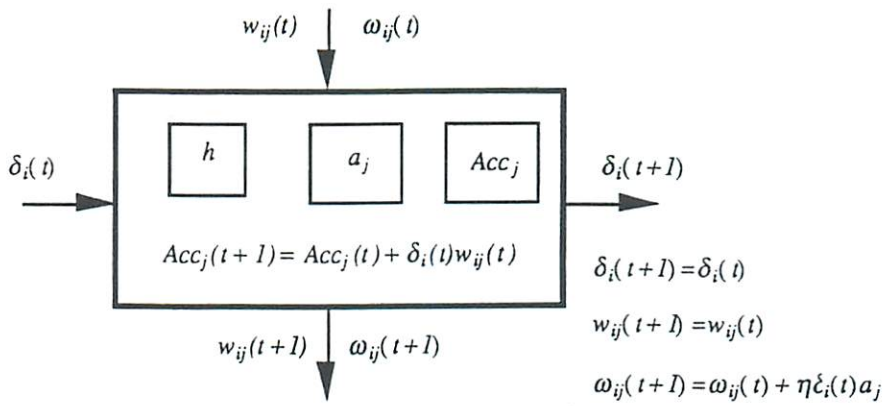


Figure 4-10 - Processing in each enabled processor of the array during the learning phase.

The processing produces modified synaptic weights ω and the error values δ for each consecutive layer of the network starting from the output down to the input layer. At each cycle the old weight value w_{ij}, received from the North port, is multiplied by the error term δ_i, received from the West port, and added to the partially accumulated sum Acc_j in each processor's local memory, see Figure 4-10. The error term δ_i is also multiplied by the neuron activation value a_j and the learning rate η, both stored in each processor's local memory, to calculate the weight modification value. The new weight value ω_{ij}, received form the North port is updated by being added the weight modification value just produced in the processor. This procedure implements the generalized delta rule learning law (as shown in Figure 4-5). The next two cycles of the data flow following the initial cycle of Figure 4-9 are illustrated in Figure 4-11. At the

completion of updating all the weight values between layer ℓ and $\ell+1$, the error term $\delta(\ell)$ for layer ℓ is calculated according to

$$\delta_i(\ell) = f'(U_i(\ell))Acc_i \quad \text{for } \ell \neq \mathrm{L} \tag{4-8}$$

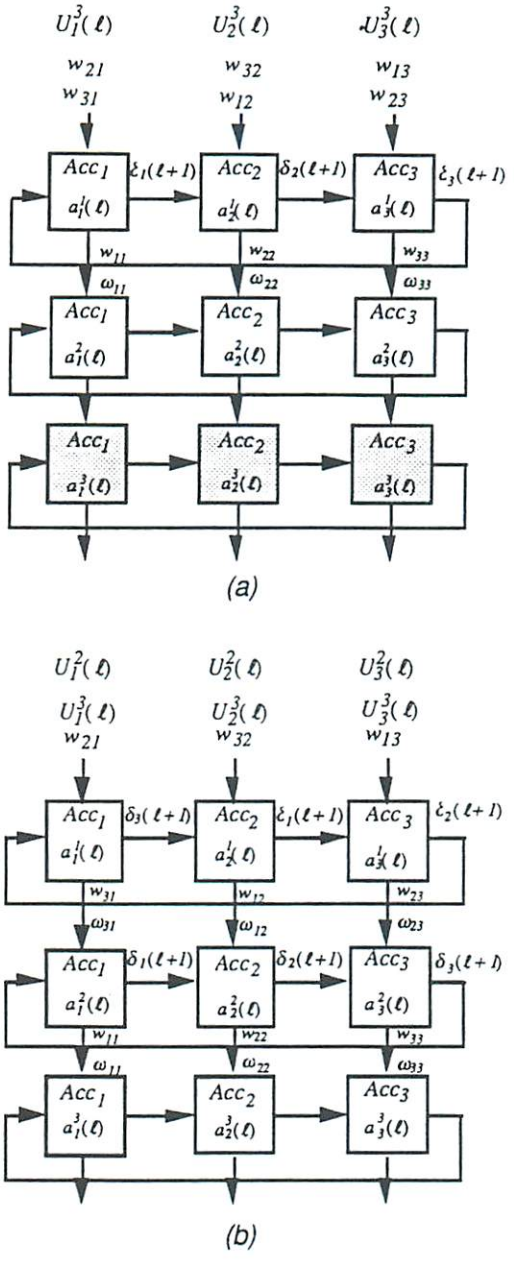


Figure 4-11 - Next two cycles of the data flow through the array after initial cycle shown in Figure 4-9. Shaded processors are disabled from performing calculations

The accumulated sum Acc_i in (4-8) is calculated concurrently while updating the weights. The term $f'(U_i(\ell))$ is calculated by receiving the partial sum $U_i(\ell)$ values from the north port and implementing the f' function in unison all the processors. The partial sum values are those that where calculated in the recall phase and stored in the global memory for use here in the learning phase.

4.3.3 Implementing Neural Networks Larger than the Processor Array Size

The neural networks used in practice have layers of different sizes and often contain larger number of neurons than there are processors in a row of a medium-grain processor array. Therefore, it is essential to address the issue of data partitioning as part of this mapping method. The partitioning in our method is implemented in such a way that the required local memory of the processors can be very small and independent of the

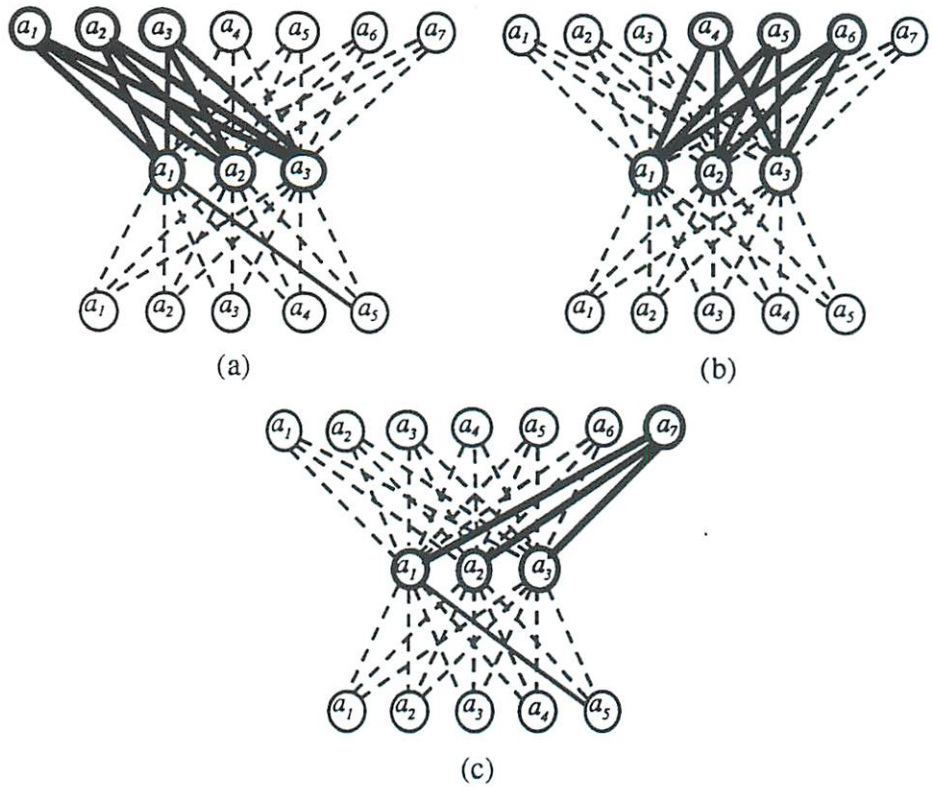


Figure 4-12 - Network partitioning for implementation on a "3 x 3" processor array

network size. If a given layer is smaller than the rows of the processor array, then some of the processors will contain phantom neurons with activation values always equal to zero. The processing in this case is not changed except that the weights between the real neurons and the phantom ones are set to zero. The layers that are larger than the rows of the processor array have to be processed in fragments of the size smaller or equal to the row size. This procedure essentially creates virtual processors through the use of time-multiplexing. The order in which each fragment is processed in the recall phase is shown in Figure 4-12.

In this example the processor array size being used is "3 x 3" (similar to those in Figures 4-6 and 4-9) and there are 7 neurons in layer 1. The activation values from the first 3 neurons on layer 1 ($a_1^i(1)$through $a_3^i(1)$, where i is the network index) are initially loaded into the processor array. The processing is performed as described earlier by propagating through the network the 9 (or 3^2) weight values corresponding to the connections between neurons $a_1^i(1)$, $a_2^i(1)$, and $a_3^i(1)$ on layer 1 and the neurons $a_1^i(2)$, $a_2^i(2)$, and $a_3^i(2)$ on layer 2, during which the partial sum values U_i's will be accumulated. The activation values for the next 3 neurons on layer 1 ($a_4^i(1)$through $a_6^i(1)$) are then loaded concurrently with the tail end of the processing phase of the last cycle, in a pipeline fashion. The next batch of 9 weight values for the connections between neurons $a_4^i(1)$, $a_5^i(1)$, and $a_6^i(1)$on layer 1 and neurons $a_1^i(2)$, $a_2^i(2)$, and $a_3^i(2)$ on layer 2 are then propagated through the processor array as before and the computation of the partial sum values continues. Finally the last neuron on layer 1 $a_7^i(1)$ is loaded into the network and processed.

After all the neurons on layer 1 have been processed in this fashion and the threshold values been added to the partial sums, the neuron activation function is applied to calculate the activation values of the 3 neurons on layer 2 ($a_1^i(2)$, $a_2^i(2)$, and $a_3^i(2)$). If there were more than 3 neurons on layer 2, the activation values just calculated for neurons $a_1^i(2)$, $a_2^i(1)$, and $a_3^i(1)$ would be unloaded to the global memory and the above process would be repeated to evaluate the activation values for the next 3 neurons on this layer. This processing is repeated until the activation values for all the neurons on the output layer have been calculated. A similar partitioning procedure is used to perform the back-propagation of the error values through the network (learning phase). In this case segmentation starts from the output layer toward the input layer, as oppose to the reverse direction in the recall phase.

The pipelined processing employed in this mapping method allows for efficient loading and unloading of activation values from the processor array in this partitioning scheme. Since these operations almost completely overlap with the emptying and loading of the processing pipeline, only a small overhead results from processing the network in small partitions.

4.3.4 Implementing Nettalk on the Hughes SCAP Architecture

As described earlier, the mapping method described in this section is suitable for machines with 2-dimensional, mesh-connected, SIMD architectures. An example of such a machine is the Systolic/Cellular coprocessor designed and developed at the Hughes Research Laboratories [62]. The architecture of this machine consists of a 16 by 16 array of processors controlled by a single controller in SIMD fashion, see Figure 4-13. The

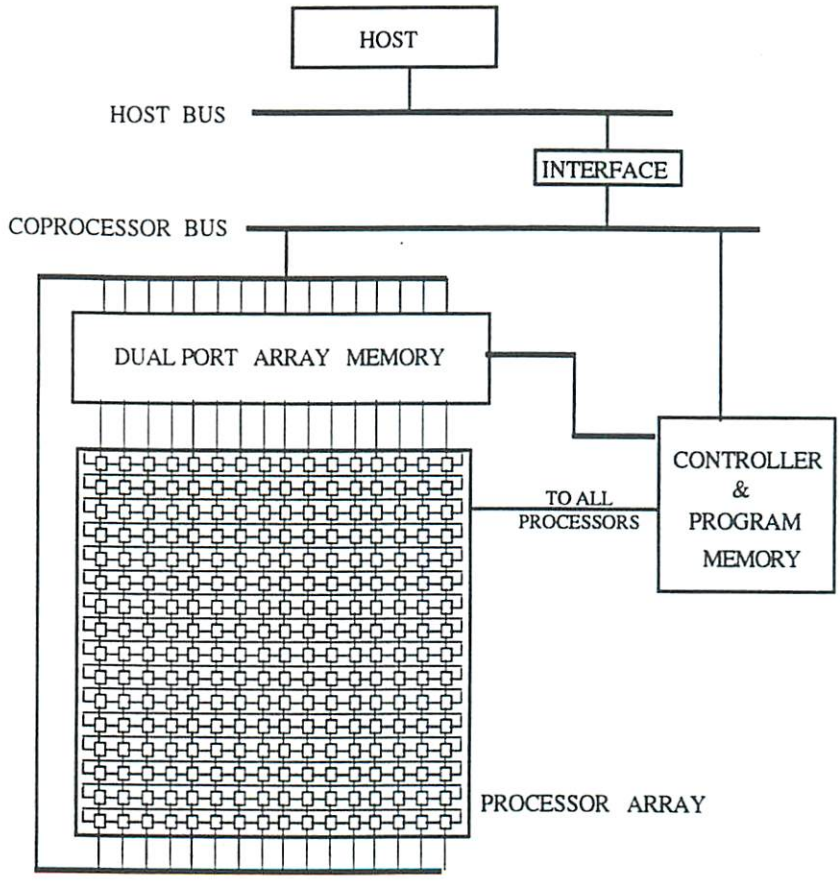


Figure 4-13 - System architecture of the Hughes Systolic/Cellular Coprocessor.

processors are connected to four of their nearest neighbors. Processors on the boundary columns are connected with each other through wrap-around connections. Each processor contains a small local memory (24 words) and seven 32-bit, fixed-point, functional units - two multipliers, two adders, a divider, and a comparator, which can all compute in parallel. A 2K words dualported memory is used as a data queue. The dual-ported data memory can be accessed in parallel by all the processors in the top and bottom rows of the array.

The mapping method was tested by implementing the well known Nettalk neural network [Sejnowski, 1987 #85] on the Systolic/Cellular machine. The Nettalk network is a good example of a working neural network, which is large enough to require partitioning for our target machine and one that has also been used by others to benchmark their implementations of neural network mappings [5, 60].

The Nettalk network is a three layer feed-forward model which learns to pronounce written English text, through the use of the back-propagation learning algorithm. The network receives seven letters as inputs and the task of the network is to generate the correct phoneme corresponding to the middle letter. The input to the network is through 203 neurons, 7 groups of 29 possible input characters. The hidden layer contains 60 neurons and the output layer contains 29 neurons. Each layer is fully connected to its neighboring layers which yields a total of 13,826 connections in the network.

In our implementation, there were 16 input patterns executing concurrently - one per row of the processor array. Two neurons were mapped into a single processor to take advantage of the multiple functional units available in each processor. For example, the *double add* and the *double multiply* instructions were used to multiply two activation values and their corresponding weights together and then added the results to the partial sum values in parallel in a single processor. It is apparent from the mapping algorithm that there are approximately equal number of computation and communication operations. This system characteristic allows for efficient and simple overlapping of computation and communication operations. In our implementation of the Nettalk network on the Hughes systolic/cellular processor, only 26.9% of the total processing time was attributed to the communication operations which were not overlapped by computation operations.

Additional parallelism was exploited by overlapping computation operations, which require more than a single instruction cycle, with communication operations. For example, a North to South data transfer in all 16 processors in a row requires 3 cycles

(when accessing global memory), and a multiplication operation takes 7 cycles. A multiplication operation was initiated in the multiplier and while this operation was being processed, the next two operands for the next instruction were fetched from the North processor concurrently. In this fashion the two communication operations did not contribute any additional time to the processing time of the implementation.

The implementation of cellular operations, which are executed in unison across all the processors in the array, are not specified in this mapping. These operations may be implemented by the most suitable method for a given target machine. A major portion of these operations are involved with the implementation of the neuron activation function. The activation function used in most back-prop implementations, including Nettalk, is the sigmoid function defined as

$$f(x) = \frac{1}{1+e^{-x}} \tag{4-9}$$

In our implementation, this function was realized using the *exp(x)* function which in turn was calculated by means of a range reduction technique [16]. This implementation of the activation function was arrived at after an extensive evaluation of other implementation methods. One such method is a table look-up scheme. This method offers fast response (one memory read instruction), but is impractical for implementation on the Systolic/Cellular machine of Hughes since it requires indirect addressing within each processor's local memory and a relatively large local memory size neither of which were available in the target machine.

4.3.5 Performance Evaluation of the Mapping Method

A measure of performance, which is becoming a standard for neural networks, is Million Connections Per Second (MCPS). This metric is used to measure the feed-forward processing rate of a given implementation for a particular neural network. Another measure, Million Connection Updates Per Second (MCUPS), is used to evaluate the performance of a given implementation for processing both the feed-forward (recall) and feed-back (learning) phases of a neural network. The execution time of a complete recall and learning cycle, including all the data loading and unloading operations, was used to arrive at a 100 MCUPS performance for our implementation of Nettalk on the Hughes

systolic/cellular system. Considering only the recall processing, we achieve a 240 MCPS performance with our implementation.

Table 4-1 combines the comparative results for this mapping method on the Systolic/Cellular Coprocessor, with mappings from [41] implemented on the same machine, and two other special purpose implementations on parallel machines - Warp [60], and Connection Machine [5], and a workstation - Sun 3/160 with a floating point accelerator as reported in [5]. The minimum local memory requirements for systems other than the Hughes systolic/cellular are estimates arrived at under the assumption that no loading/unloading operations from an external memory are allowed. The performance of the Connection machine implementation was also estimated using the performance reported in [[5] multiplied by two to account for expected improvements due to code optimization. The implementation on the Warp used 32 bit floating point data with a floating point adder and multiplier and an integer ALU. Implementation on the Connection Machine used a two bit serial ALU per processor. The data type used in our implementation was 32 bit integer with integer arithmetic functional units.

	SYS/CELL HRL	SYS/CELL S.Y. KUNG	WARP	CM1 - 16K	SUN 3/160 + F.P.A.
PROCESSING RATE (MCPS)	100	18	17	7	0.034
MINIMUM LOCAL MEMORY (WORDS)	10	29	6182	5	14000

Table 4-1 - Comparison of different implementations of the Nettalk neural network based on throughput and local memory size requirements.

4.3.6 Applicability of the Mapping Method

The execution procedure described in this section could be applied to many layered feed forward neural networks. Even though this mapping method uses a SIMD architecture, the activation functions used for the neurons can vary from layer to layer. Moreover, within each layer there is also a limited variability possible. For example if a polynomial approximation is being used to compute the neuron activation function, different

polynomials can be implemented in each neuron at the same time by having different coefficients stored in each processor's local memory. If the network requires significantly different activation functions in the same layer, it can be accomplished by sequentially applying the desired functions and disabling/enabling appropriate neurons. Since the learning rate η is stored in the local memory of each processor, different neurons in the network could, if desired, use different learning rates with no effect on the performance of our mapping.

Neural networks with feedback architectures (such as the Bidirectional Associative Memory [38] and the Hopfield net [33]) can be implemented with the same type of architecture and style of mapping. The basic computation involved in the processing of these networks can be represented as matrix and vector calculations as described in Chapter 2. The data organization and movement through the array of our mapping implements these operations very efficiently. The cellular operations in the algorithm, such as the activation function evaluation, could be changed to any other type of cellular operations without any effect on the inter-processor communications. This allows for great flexibility in the models that can be implemented with this type of mapping.

4.3.7 Deficiencies and Shortcomings of the Mapping Method

Although this mapping method can achieve very high processing speeds, as demonstrated in Section 4.3.5, it nevertheless has similar shortcomings to the mapping on fixed-size 1-D ring architectures mentioned in Section 4.2.4. Namely, the mapping requires a good match between the number of columns in the processing array and the number of neurons per layer of the neural network. In addition, although the processing array is organized in a 2-D topology, the mapping method treats the processors as a pipeline of 1-D ring processors. This type of mapping cannot take advantage of the 2-D nature of the processor array when manipulating 2-D data structures, such as images.

Because of these limitations, the DREAM Machine architecture has been proposed and various mapping methods have been developed to remedy these problems. The mapping methods are described in detail in the following two chapters.

Chapter 5

The Algorithmic Mapping Method

Mapping methods for implementing regularly structured neural network models can take advantage of the regularity in their interconnection structures and devise a systematic and direct approach for solving the assignment and scheduling problems mentioned in Section 4.1.2. In this chapter we will describe the use of such an algorithmic approach used to arrive at efficient mappings of regularly connected neural networks with dense interconnection structures onto the DREAM Machine architecture. This mapping method is based on the processing principles described in Section 4.1 and can be considered as an extension of the mapping method for implementations on fixed-size ring architectures described in Sections 4.2 and 4.3.

5.1 Applicability of the Algorithmic Method

As mentioned previously in Chapter 4, a major problem with implementations on 1-D ring architectures is attributed to the requirement of having equal number of neurons and processors in the ring to achieve maximum efficiency. The flexibility in the communication network of the DREAM Machine architecture allows for embedding of variable length 1-D rings on the 2-D processor array topology of the machine [71]. Therefore, the size of the processing ring can be varied to match the number of processors in the ring to the number of neurons in the specific layer of the neural network being processed. With such an approach, fine grain implementations with large number of processors can be utilized to implement any size neural networks. Furthermore, neural network models comprised of multiple densely connected blocks of neurons, with

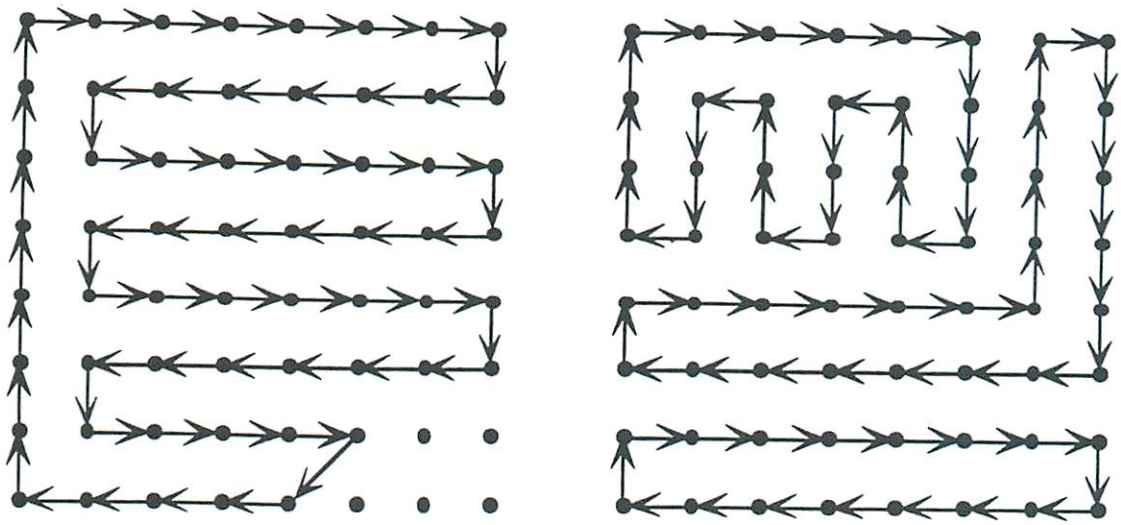
limited interconnections between various neurons can be implemented efficiently by concurrently processing multiple processing rings on the machine.

In summery, the mapping method is applicable to neural network structures with single or multiple layers of neurons where each layer can be comprised of one or more blocks. The mapping method produces the necessary assignment of neurons to processors and establishes the associated computational paths through the processing array. This is accomplished by employing a processing ring approach, similar to the one described in Chapter 4, for linearly assigning neurons to PEs of the ring and scheduling the paths as complete traversals of the processing ring.

5.2 Implementation of Variable-Size Processing Rings on the DREAM Machine

With the ability to vary the ring size to match the neural network size, the system throughput of the DREAM Machine can be maintained at a high level compared to that of the fixed-size ring array. With a processing array size of P processors, a ring of size R can simply be embedded onto the 2-D interconnection topology by folding the ring into a "snake-like" shape, see Figure 5-1. This is accomplished by setting the X-net switches of each processor in the DREAM Machine in a specific configuration as required by the folding. Figure 5-1(a) shows how a ring topology can be created on the DREAM Machine architecture with the ring length less than the number of PEs. Three disjoint rings of differing lengths are depicted in Figure 5-1(b). Due to the SIMD nature of the processing employed by the DREAM Machine, in cases where multiple rings of varying sizes are implemented concurrently, processors assigned to a specific ring are enabled only during the cycles required to complete the ring traversal. Thus, in such as scheme, the number of cycles associated with processing all the rings is determined by the length of the longest processing ring.

In cases where the neural network size exceed the number of processors in the processing array, time multiplexing techniques can be employed, as before, to create virtual PEs. A further advantage of the variable-size ring mapping method can be demonstrated by closely examining the effect of time-multiplexing the processors on the system efficiency. Creation of virtual processors on the fixed-size ring architecture



(a) (b)

Figure 5-1 - Using the reconfigurable switches of the DREAM Machine to construct circular rings on the processing array. (a) A single ring of size 59. (b) Three disjoint rings of different sizes executed in parallel.

involves assigning multiple neurons to different time slices of a single processor. For example, if a neural network with 257 neurons in a particular layer is to be processed on a processing array with 256 processors, two consecutive time slices of each processor must be used to create a ring virtual ring of size 512 PEs. Therefore, the execution rate of the implementation will correspondingly be $O(512)$ requiring to 255 unnecessary operations. On the other hand, with the variable-size ring approach of the DREAM Machine, a virtual ring of size 258 PEs can be constructed by using two consecutive cycles of each processor in a 128 PE long ring. This mapping only requires one unnecessary operation due to the round-off factor and yields an execution rate of $O(128)$.

5.3 Using Variable Length Rings to Process a Single Layer

The ability to construct variable length rings on the 2-D processing array can be efficiently utilized for neural network processing. In Chapter 2 we described the basic mapping method for implementing neural networks on 1-D ring connected architectures. We

further mentioned how the mismatch between the number of processors and the number of neurons can lead to inefficient processing. In this section we analyze the performance of the mapping method given the ability to adjust the processing ring length.

First let us discuss the use of variable length processing rings for implementing one and two layer neural networks. The extensions to this approach for implementing multilayer and blocked structured networks are presented later in this chapter. Let P represent the total number of processors in the machine and lets define R_ℓ to be the number of PEs assigned to a specific processing ring used for computing the neuron output values of layer ℓ. For example, in Figure 5-1(a) P=64 and R=59. We further define v_ℓ as the number of output layer neurons assigned to each PE in the processing ring of length R_ℓ, and ω_ℓ as the number of input layer neurons assigned to each PE of the ring. In the case of one layer neural networks, such as the Hopfield net [33], all the neurons are treated as belonging to both the output and the input layers. Formally, v_ℓ and ω_ℓ are defined as

$$v_\ell = \left\lceil \frac{N_\ell}{R_\ell} \right\rceil \qquad \text{and} \tag{5-1}$$

$$\omega_\ell = \left\lceil \frac{N_{\ell-1}}{R_\ell} \right\rceil. \tag{5-2}$$

The v_ℓ and ω_ℓ values are used to address the need for time-multiplexing multiple neurons on a single processor. In general, if the number of processors in the machine is larger than the number of neurons, both v_ℓ and ω_ℓ will have a value of one and thus their multiplicative effects can be removed from our equations. Using our notation, equation (4-5) which is used to calculate the time required for evaluating the output values of neurons in layer ℓ can be rewritten for the variable-size ring implementation as

$$T_\ell = \left(v_\ell \omega_\ell R_\ell k_1 + v_\ell k_2\right), \quad \text{where } 1 \le R_\ell \le P. \tag{5-3}$$

It can be noticed that the execution time is no longer a function of the processor array size, rather it is now dependent only on the number of processors used in the ring and the number of neurons assigned to each processor of the ring. For the sake of simplicity, in our examples in this chapter, unless otherwise specified, we assume that $N_\ell \le R_\ell$ and $N_{\ell-1} \le R_\ell$ so that $v_\ell = 1$ and $\omega_\ell = 1$.

In general, for neural network structures with no regular sparse structure between two layers of neurons, we can determine the appropriate ring length as

$$R_\ell = \left\lceil \frac{N_\ell}{\lceil N_\ell / P \rceil} \right\rceil \tag{5-4}$$

This equation takes into account the need for time-multiplexing multiple neurons on a single processor when the neural network size is larger than the machine size. Figure 5-2 shows a graph of R_ℓ as a function of the number of neurons for a fixed processor array size of 256 PEs. It can be noticed that as the ratio of number of neurons to the number of processors increases, R_ℓ asymptotically approaches P. Formally,

$$\frac{N_\ell}{P} >> 1 \quad \Rightarrow \quad R_\ell \rightarrow P. \tag{5-5}$$

This indicates that in order to benefit from the variable length ring feature of the architecture, the machine size should be on the same order as the neural network size. Consequently, this architecture favors fine-grain implementations with large number of processors.

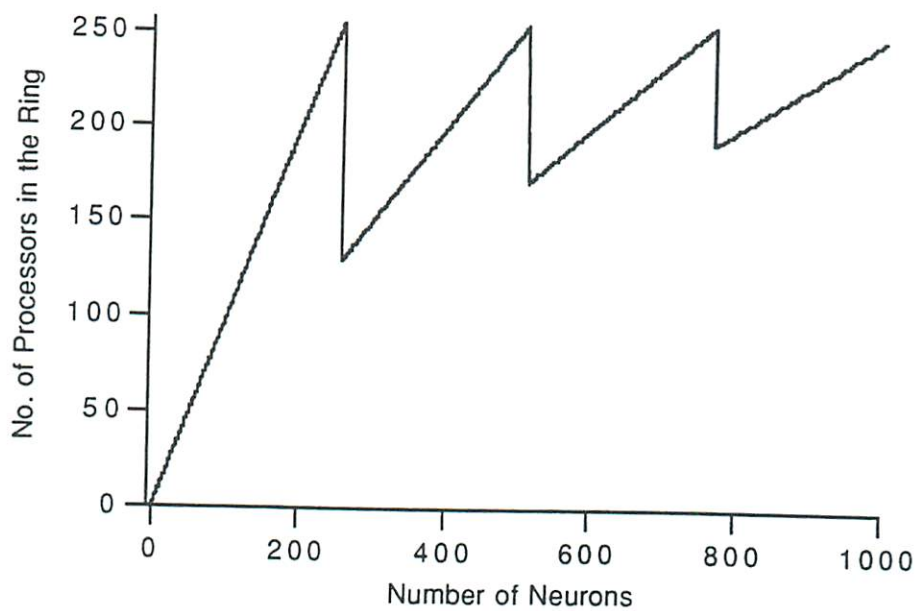


Figure 5-2 - Plot of R_ℓ vs. N_ℓ with P=256 processors.

As mentioned previously in Section 4.2.4, the mapping on fixed-size ring architectures can cause great inefficiencies if the number of processors in the machine is considerably larger than the neural network size. Thus, such mappings are more

applicable to medium-grain architectures where the inefficiency in implementing neural networks smaller than the machine size and the rather limited speedup gained due to lack of utilization of all the available parallelism in large neural networks is some what balanced. Figure 5-3 shows a plot of the execution rate of both fixed-size ring and variable-size ring implementations. It can be observed that as long as the number of neurons is smaller than the machine size, the variable length ring approach maintains a performance close to optimal, whereas the fixed-size ring approach achieves a worst-case constant execution rate.

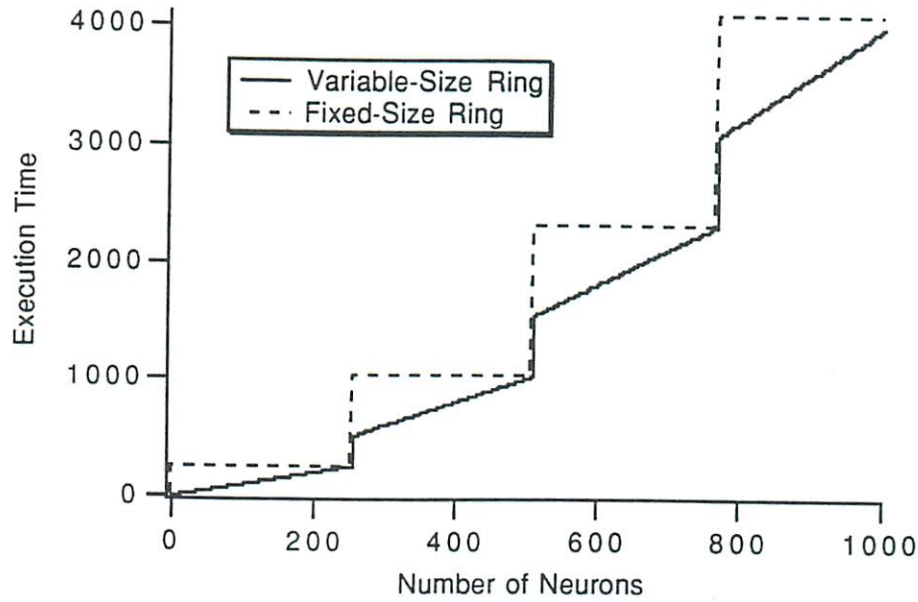


Figure 5-3 - Plot of T_t vs. N_t for the fixed-size ring mapping defined by equation (4-5), and for the variable-size ring mapping defined by equation (5-3). The parameters k_1 and k_2 are assumed to be 1 in order to simplify the comparison.

5.4 Implementing Multilayer Neural Networks

For the one and two layer neural networks with no regular block interconnection structure (e.g., the Hopfield net [32] and the BAM network [38]), a single ring is sufficient to efficiently implement these networks. In case of multilayer neural networks (e.g. the Multilayer Perceptron model [66]), where neuron output values are propagated through consecutive layers of the network, a more complicated ring structure can be designed

which dynamically matches the number of PEs in the ring to the number of neurons in each layer of the network being processed. This is accomplished by dynamically changing the ring size as the computation moves from layer to layer with $R_\ell = \max\{N_\ell, N_{\ell-1}\}$ for each layer ℓ. As stated earlier, for the sake of simplicity we assume that there are enough processors in the system so that $N_\ell \leq P$ and $N_{\ell-1} \leq P$.

As an example, a three layer neural network with $N_1 > N_2 > N_3$ (see Figure 5-4), can be mapped efficiently to a ring structure of length N_2 embedded in a longer ring of length N_1, as shown in Figure 5-5. In the first N_1 cycles of processing the partial sum values u's are propagated through the large loop of length N_1. At the completion of this phase, the final u values for the N_2 neurons in the second layer are available in the first N_2 PEs of the ring. All the processors can then perform the application of the neuron activation function in unison to evaluate the second layer neuron's output values. Since the large ring of length N_1 is folded in such a way that PE #1 and PE #N_2 are adjacent to each other, the communication switch between these PEs can be reconfigured to implement a ring of size N_2, see Figure 5-5.

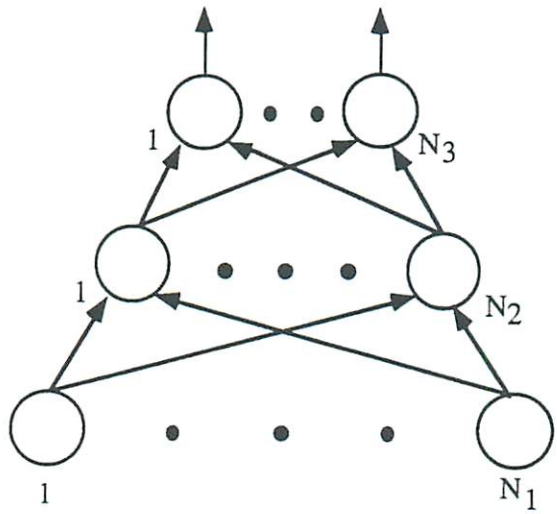


Figure 5-4 - A three layer neural network with $N_1 > N_2 > N_3$.

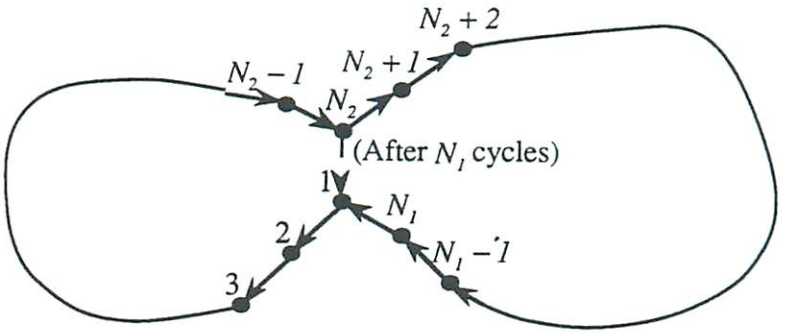


Figure 5-5 - An embedded ring structure containing a ring of size N_2 embedded in another ring of size N_1.

After N_2 processing cycles, the first N_3 PEs in this ring will have the final u values of the neurons in the output layer. After the application of the transfer function, the network output values are ready to be stored in memory for further processing or accessed by the host computer. The total time required to complete the computation of a network with L layers is

$$T = \sum_{\ell=1}^{L} T_\ell , \tag{5-6}$$

with $\nu_\ell = 1$ and $\omega_\ell = 1$ the total time to complete the network computation can be calculated using equation (5-3) as

$$T = k_1 \sum_{\ell=2}^{L} R_\ell + (L-1)k_2 . \tag{5-7}$$

In cases where the number of neurons in one layer is less than the number of neurons in the next higher layer ($N_{\ell-1} < N_\ell$), a similar ring embedding structure can be created. In such cases, layer ℓ-1 can be padded with zero-valued neurons in order to make $N_{\ell-1} = N_\ell$. This results in the creation of only a single ring of size N_ℓ PEs. In general, the ring length needed to implement the processing associated with layer ℓ is

$$R_\ell = \max(N_\ell, N_{\ell-1}). \tag{5-8}$$

By means of this ring embedding technique, various ring structures can be devised for efficient implementation of each layer in the neural network. A number of different neural network structures and their associated processing rings used for their implementation are illustrated in Section 5.10.

The case where the number of neurons exceed the number of processors can be addressed through the use of time-multiplexing multiple neurons on a single PE. The number of time slices required for implementing each individual neuron on a single PE is designated by ν_ℓ and ω_ℓ defined in (5-1) and (5-2). In multilayer processing the outputs of layer ℓ-1 are used as inputs to layer ℓ. Thus if $\nu_{\ell-1}$ neurons of layer ℓ-1 are mapped to each PE, there should be that many neurons per processor for inputs to layer ℓ. In other words, $\omega_\ell = \nu_{\ell-1}$. This restriction disqualifies the simple use of equations (5-1) and (5-2) in calculating ν_ℓ and ω_ℓ for all the layers when the neural network size exceed the processing array size. Due to the non-linear nature of equation (5-3) used to calculate the execution rate of the mapping, determining the optimum values for ν_ℓ and ω_ℓ is non-trivial and require a heuristic search method.

5.5 Implementing Back-Propagation Algorithms

Some neural network learning algorithms, such as error back-propagation [66], require that the error values be calculated and propagated from the output layer back to the input layer after the evaluation of the output layer neurons. During the backward propagation of error values, the synaptic weights between the neurons of consecutive layers are adjusted according to the specific learning algorithm being employed. The computational aspects of the most commonly used learning algorithm, the back-prop algorithm [66], was described in Section 4.3.1. This computation can be implemented using the variable-size ring mapping method by keeping the partial sum values associated in the calculation of the error contributions (Acc_i in Figure 4-5) and the neuron activation values a_i local in each PE while rotating the error terms δ_i through the ring structure. This approach is similar to that described in Section 4.3.2 and [Shams, 1991 #105] except that the ring length can now be adjusted to fit the neural network size for increased efficiency and throughput.

In this approach, the construction and traversal of the embedded ring structure is performed in the reverse order. In other words, the ring of size R_ℓ is traversed followed by the ring of size $R_{\ell-1}$ and so on. This procedure is continued until processing for layer 2 is completed and all the weights in the neural network have been modified. In this mapping, a complete network updating operation is executed before the next one is initiated. Other mapping methods [72, 89] take advantage of network level parallelism

and execute multiple training patterns in parallel. The use of this approach is described in detail in Section 5.8.

The execution rate for implementing the backward error propagation algorithm has the same order of complexity as that of the forward neural computation described by equation (5-3). However during each systolic cycle of the algorithm, the number of algebraic operations is increased by two multiplication and one addition operations. Similarly, the time required to implement the neuron transfer function is replaced by the time required to implement the first derivative of this function. If the sigmoid function, equation (4-9), is used as the neuron transfer function, the computation of its first derivative can be simply performed as

$$f'(U_i) = a_i(1 - a_i), \tag{5-9}$$

which requires only one subtraction and one multiplication operations. Such a scheme is only useful if the architecture allows for enough local memory area to store all the activation values for the neurons of consecutive layers mapped to a single PE.

5.6 Implementing Block Connected Neural Networks

In addition to the layered interconnection structures, a number of neural network models use or allow the use of blocked connected interconnections. In our treatment here, a block consists of two disjoint sets of neurons with full interconnections between the sets and no connections within neurons of each set. Such interconnection structures are more general than the layered networks since each layer can be represented as being comprised of a single block. A more complex structure might utilize several blocks within each layer. We can treat the mapping method described above as a single block per layer case and extend this mapping to multiple blocks per layer structures.

Block structures are commonly used for two reasons. First, block structured networks can be used to perform data fusion by combining the outputs of several disjoint portions of a neural network. An example of such a structure is shown in Figure 5-6. The second type of block structure is usually employed to perform some type of feature detection using a concept called weight sharing [46]. These networks generally perform a convolution type operation on the previous layer's néurons using the synaptic weights as a mask filter. The interconnection structure of these networks contains many small and

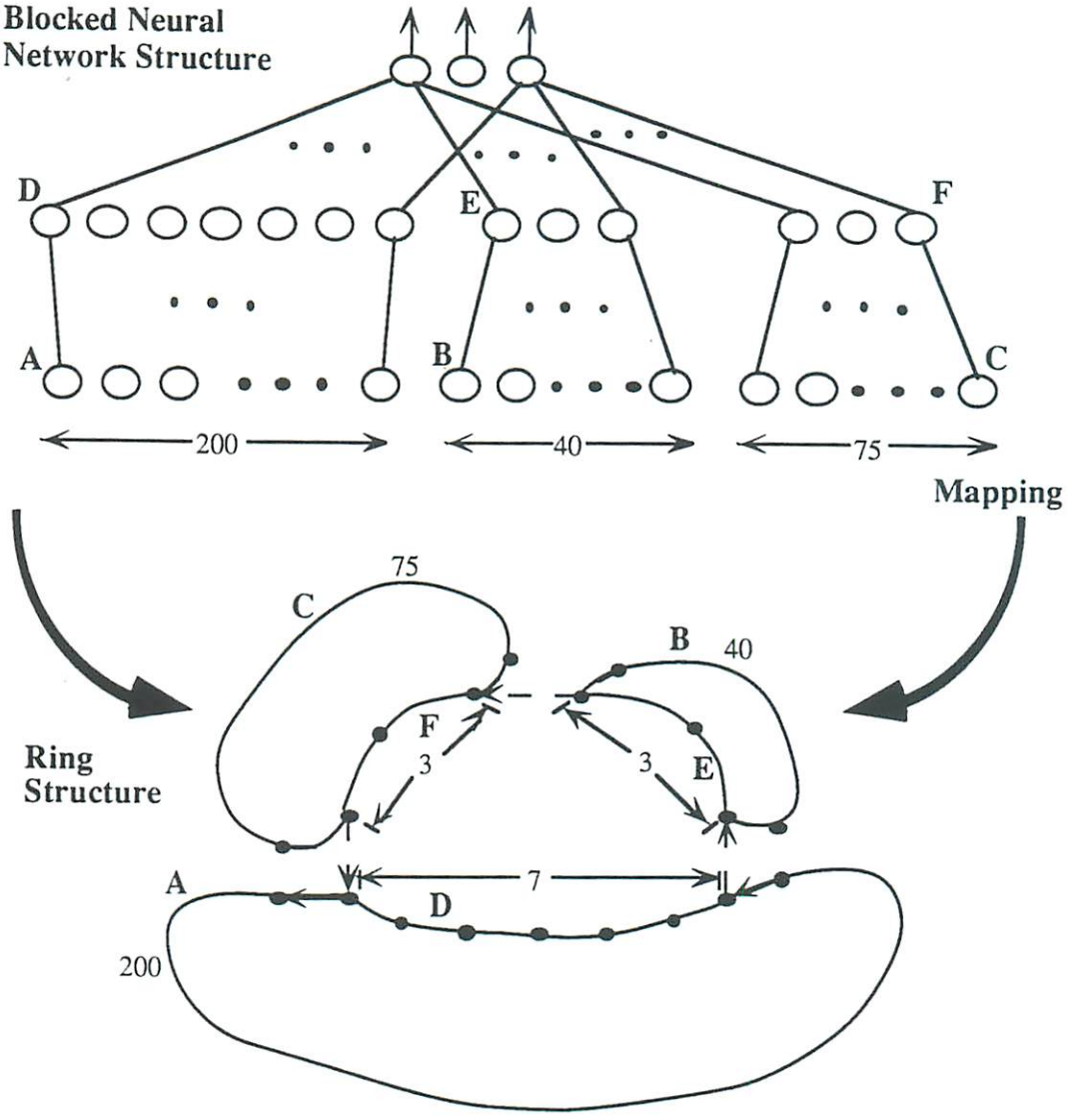


Figure 5-6 - A blocked structured neural network utilizing 3 disjoint blocks between the input and hidden layers and its associated mapping on 3 processing rings.

overlapping blocks. An example of such a network is the neocognitron model used for invariant object recognition [19]. In this section we describe the mapping methods best suited for implementing each style of interconnection structure on the DREAM Machine using the variable length ring processing scheme.

5.6.1 Implementing Data-Fusion Style Neural Networks

The simplest case of implementing block connected networks is when each block is completely disjoint from other blocks in the network. In this case ring structures such as those described earlier in this section can be used to process each block on different parts of the processing array. This type of mapping leads to ring structures similar to that shown in Figure 5-1(b). If the blocked structure network is from the class of models used for data fusion, the output of a number of blocks is combined and used as inputs to a single block and possibly processed by additional layers in the neural network (such as the one depicted in Figure 5-6). Lets define N_ℓ^b as the number of neurons in block b of layer ℓ, and R_ℓ^b as the length of the processor ring associated with that block. If the output of several blocks of layer ℓ are to be treated as input to layer $\ell+1$, processing rings associated with each block of layer ℓ are placed next to one another to form another ring for computation associated with layer $\ell+1$. For example, in Figure 5-6 after 200 cycles used for completion of the longest ring, the switch setting between the PEs of adjacent rings will be changed to form a processing ring of size 13. After 13 computation cycles in this configuration, the final neuron values for the output neurons are available in the first three PEs in this ring.

The number of cycles required to implement one layer of a blocked connected neural network using this mapping method is

$$T_\ell = \left(\tilde{v}_\ell \tilde{\omega}_\ell \tilde{R}_\ell k_1 + \tilde{v}_\ell k_2\right), \quad \text{where} \tag{5-10}$$

$$\tilde{v}_\ell = \max_{b \in B_\ell}\left\{v_\ell^b\right\}, \quad \tilde{\omega}_\ell = \max_{b \in B_\ell}\left\{\omega_\ell^b\right\}, \text{ and } \tilde{R}_\ell = \max_{b \in B_\ell}\left\{R_\ell^b\right\}. \tag{5-11}$$

In equation (14), B_ℓ represents the set of all blocks that comprise layer ℓ of the network. Since multiple blocks are executed concurrently on different processing rings of the machine, the time required to complete the computation associate with all the blocks is equal to the time required to complete the longest ring. Thus, $\tilde{R}_\ell$ is set equal to the longest ring of the layer. The DREAM Machine's masking capability is used to inhibit

the processors in all the rings that have completed their computation before the longest ring is completely traversed. The number of neurons in layer ℓ assigned to each PE of the ring must be equal, or treated as being equal, for all the different blocks in that layer. This is due to the SIMD execution paradigm used by this mapping. Therefore, $\tilde{\nu}_\ell$ and $\tilde{\omega}_\ell$ are set equal to the largest value of ν_ℓ and ω_ℓ, respectively, of all the blocks in layer ℓ, as represented by in equations (5-11).

5.6.2 Implementing Feature-Detection Style Neural Networks

A different approach can be taken when implementing block connected structures with overlapping blocks of neurons on the DREAM Machine. In many cases involving structures with overlapping blocks of neurons, a weight sharing technique is used to create specific feature detecting neurons [19, 45]. Weight sharing is a concept where all the neurons have input synapses of the same spatial distribution. In other words, all neurons share the same input synapse values connected to different overlapping blocks of neurons. Efficient implementation of such structures can be accomplished by mapping each unique synaptic weight value to a specific PE and storing the corresponding neuron activation values in the local memory of each processor. In this fashion there is only a single copy of each synaptic weight stored in the processing array and only a few redundant copies of the neuron activation values are stored in the array, depending on the amount of overlap between adjacent blocks. This mapping approach is more efficient when the number of processors in the processing array is close to the number of unique weight values in the neural network.

5.7 Implementing Neural Networks Larger than the Processor Array

In the above discussion we assumed that there is always enough processors to constructs rings of arbitrary size R_ℓ. This condition can be satisfied if the number of neurons in the largest layer of the neural network is less than or equal to the processor array size. To formulate the above mapping technique in a more general fashion we employ the use of time-multiplexing in creating *virtual* PEs. The number of virtual PEs that can be implemented is an integer multiple of the processing array size. The

formalism given in equations (5-3) and (5-10), for arriving at the execution rate of this mapping, incorporates the effects of the required virtual PEs through the use of the v_t and ω_t terms.

The goal of an optimal mapping will be to determine the appropriate values for the R_t values such that the total execution time (given by equation (5-7)) is minimized. Due to the noncontinuities in the execution time equations introduced by the ceiling functions, we cannot directly solve for the best R_t values. In many cases the network structure can give good hints as to approximate values for the ring sizes. Since the execution rate of a specific mapping can be evaluated analytically using equation (5-7), optimization techniques, such as simulated annealing [36], can therefore be utilized to arrive at efficient mapping solutions.

5.8 Batch-Mode Implementation

As described earlier, batch-mode processing utilized network level parallelism by simultaneously implementing multiple instantiate of a single neural network on a parallel processor. This type of mapping has been extensively used to increase throughput rates for a number of different parallel implementations [49, 60, 72, 88, 89]. Batch-mode processing can be used to improve the system utilization of the mapping method in cases where the number of neurons in the network is smaller than the number of processors in the processor array. The DREAM Machine's processor array can be configured into many small regions of size equal to the size of the neural network being implemented. Each of these regions can independently implement the complete network assuming that input patterns associated with each network are stored locally in each region. This type of processing requires that a number of different input patterns be available to the system before the processing is initiated.

Batch-mode processing can also be used to implement neural network learning algorithms. There are two major disadvantages with using batch-mode learning. The first is due to the lack of models that allow for batch-mode learning. The second problem is associated with the effectiveness of this training approach. In gradient descent based learning algorithms, such as back-propagation [66], the mathematically correct algorithm requires weight updates after each pattern presentation. By using batch-mode learning,

true gradient descent is not implemented since weight values are updated according to the sum of the weight updates associated with a number of different training patterns. Another attribute of this approach is that by combining several weight contributions together, the learning process might be slowed proportional to the batch size. Therefore, the speed gained by running multiple networks in parallel is lost by having to increase the number of learning cycles by a factor close the batch'size. In Section 5.10 we compare a number of different neural network implementations based on their throughput. Although a considerable amount of increased utilization and throughput can be obtained, in these comparison we will not consider the use of batch-mode processing due to the problems mentioned here.

5.9 Implementing Competitive Learning

Until now we have assumed a neuron transfer function that can be implemented based on data stored local to each processor, such as the sigmoid function and the threshold function (see Figure 2-2). In this section we examine methods for efficiently implementing neural network models utilizing competitive learning algorithm on the DREAM Machine. These algorithms update the weight values associated with synaptic connections arriving at the inputs of the neuron with the largest activation value, called the winner neuron, in the specific layer. Thus the transfer function associated with the neurons in these models depend on activation values of a number of other neurons scattered in the processing array. Certain models, such as the self-organizing feature maps [37], modify the weight values of neurons in a local neighborhood of the winning neuron in addition to the connections arriving at the winning neuron.

In Chapter 3 we described how the G-bus of the DREAM Machine architecture can be utilized to efficiently determine the winning neuron. Using the conditional masking instruction, all the PEs except the one assigned the winning neuron can be masked out from executing the weight modification procedure. In our mapping the weight values associated with the inputs of the winning neuron are distributed across the different PEs in the ring. Nevertheless the learning algorithm can be implemented by rotating the mask bit through the PEs of the ring starting from the winner PE. In this fashion the appropriate weight associated with the winning neuron is modified in the properly

enabled processor. This procedure requires $O(R_t)$ time steps, where R_t is the number of neurons in the ring being evaluated.

A more efficient weight updating method can be devised using the local address modification capability of the DREAM Machine (described in Section 3.3) in addition to the G-bus. In this approach, the local memory of each processor holds the neuron ID value indicating the neuron's relative location in the ring. The input weights associated with neuron i are stored in memory location (*W_Base+offset*) of the processor assigned neuron j, where *W_Base* represents the base address of the weight memory space and *offset* is calculated as

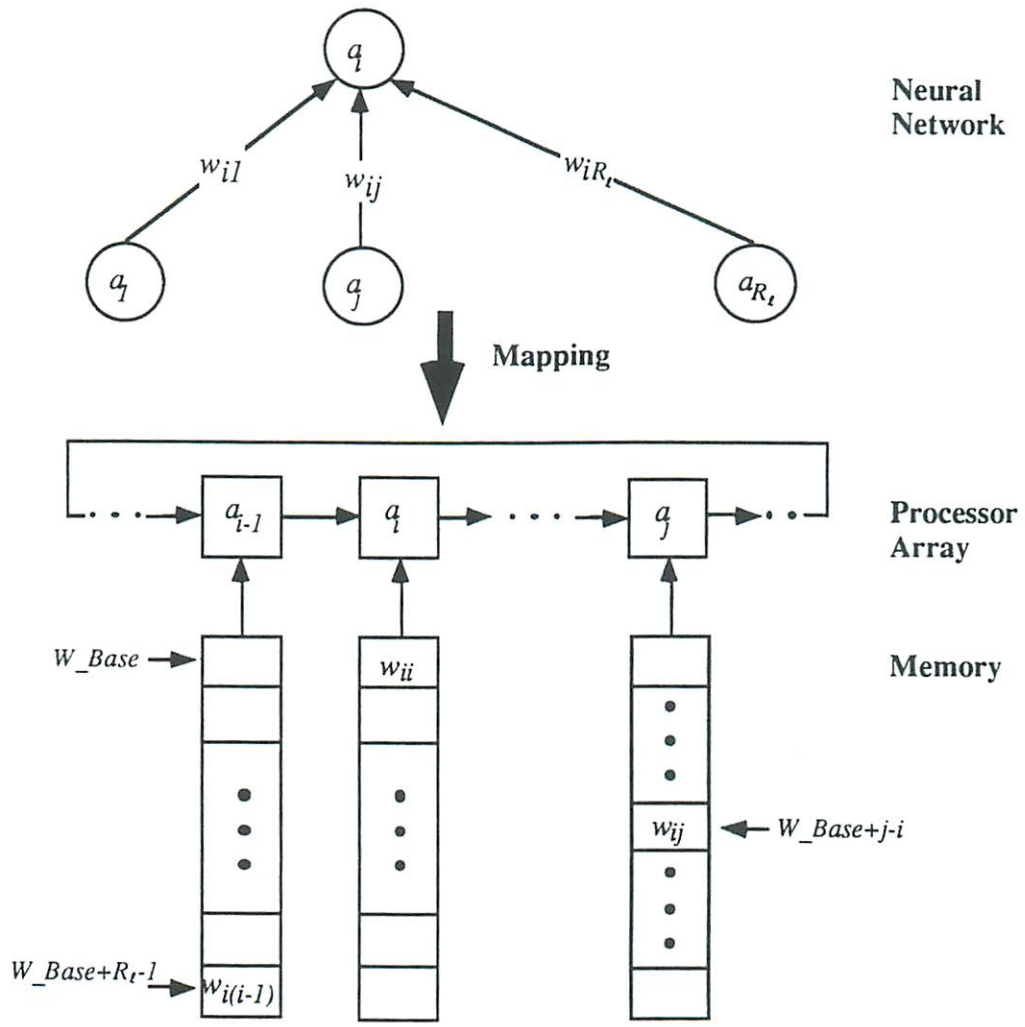


Figure 5-7 - Memory locations of synaptic weights on the inputs of neuron i.

$$offset = \begin{cases} j - i & \text{if} \quad j \geq i \\ R_t + j - i & \text{if} \quad j < i \end{cases} , \qquad (5\text{-}12)$$

see Figure 5-7.

In order to achieve maximum parallelism in the weight updating phase, after the determination of the winning neuron, the locally stored neuron ID value (ID_{max}) is broadcasted to all the PEs in the machine by the controller using the Instruction/Data broadcast mechanism described in Section 3.2. All the processors can then calculate their corresponding memory offset values according to equation (5-12). The controller can then load each PE's local address register with *W_Base*, and shift the offset value into this register in $O(Log\ R_t)$ time steps. At this point all processors can access the appropriate weight values and perform the update function in unison. For large values of R_t, this approach yields a higher performance requiring only $O(Log\ R_t)$ steps compared to the earlier mention method which yields a performance of $O(R_t)$ steps.

5.10 Implementation Examples and Performance Evaluation

In this section we demonstrate the performance of the DREAM Machine through the use of several "real-world" example mappings. We compare this result with that obtained from mappings onto a ring systolic architecture. Due to the nature of the mapping method, other more complex interconnection topologies (e.g. hypercube, plain mesh, etc.) do not offer additional advantages over the 1-D ring and thus are not included in this evaluation.

5.10.1 Performance Metric

A commonly used and quoted measure of implementation performance for neural networks is the Million Connection Updates Per Second (MCUPS) metric. This measure is calculated by dividing the total number of connections in the neural network by the amount of time required to perform a weight updating procedure on the complete network. The execution time is measured starting from the point where input data is presented to the network through the point where all synaptic weight updates have been

completed. The main problem with using this measure as a performance metric is that the weight values can be updated using a number of different learning algorithms each having a different amount of computational complexity.

For evaluating and comparing the performance of our implementation we use a less restricting metric based on recall, or feed-forward, processing only. The metric used here is the Million Connections Per Second (MCPS). This measure is evaluated by dividing the number of connections in the neural network by the total time between the presentation of the input pattern until all the output values are generated. Use of "real-world" example networks offers a better assessment of the system's ability in efficiently implementing varying network structures over the often quoted peak execution rate.

Optimally, the number of operations that can be performed in parallel with any implementation taking advantage of neuron level parallelism (assigning individual neurons to distinct PEs) is equal to $\min(N_\ell, N_{\ell-1})$ for processing layer ℓ. The optimum execution rate for implementing the processing associated with layer ℓ can be represented as

$$T_\ell^* = \frac{N_\ell N_{\ell-1}}{\min(N_\ell, N_{\ell-1})} = \max(N_\ell, N_{\ell-1}). \qquad (5\text{-}13)$$

A relative measure of performance of any mapping method can be evaluated as a ratio of the mapping's execution rate over the optimal. Three different neural network structures have been selected as benchmarks for evaluating our mapping method applied to the DREAM Machine architecture. We use both the MCPS and the percentage of optimality measure in the evaluations.

5.10.2 Implementing a Fully Connected Multilayer Neural Network

A commonly used neural network model for comparing relative performances of parallel implementations of neural network is the Nettalk network [69]. Nettalk is a simple three layer neural network with full interconnections between adjacent layers which utilizes the back-propagation learning algorithm to produce speech from printed text. The network consists of 203 input neurons, 60 hidden neurons (neurons in the second layer), and 29 output neurons. Using the algorithmic mapping method describe earlier in this chapter, we can construct a ring structure (similar to Figure 5-5) with N_1=203 and N_2=60.

Assuming a DREAM Machine architecture with 256 processing elements, the implementation execution rate can be arrived at by using equations (5-3) and (5-6) to be $T=263k_1+2k_2$ seconds, where k_1 is the time to execute a single systolic cycle of the algorithm and k_2 is the time required to perform the table lookup operation.

Since the DREAM Machine architecture allows for memory access, interprocessor data transfer, and arithmetic operations, to be executed in parallel, k_1 will be equal to the time required to implement the most time consuming of these three operations. Assuming current technology for the implementation of the DREAM Machine, we can safely assume a value of k_1=100ns. Assuming an 8-bit quantization level for the neuron activation function, the time required to implement the table lookup operation will by 9 memory access cycles (8 for loading the shift register and 1 for reading the final value). Allowing for a 50ns memory access time, k_2=450ns.

Using these values we arrive at a performance measure of 512 MCPS for implementing the Nettalk network on the DREAM Machine. The implementation of the same neural network model on a fixed ring systolic array, following the mapping in [41], with a ring size of 256 PEs, will achieve a throughput rate of $512k_1+2k_2$ seconds. Allowing for the same implementation technology and setting k_1=100ns and k_2=450ns, we arrive at a 267 MCPS throughput rate. This assumes that the ring architecture supports the same type of table lookup mechanism as the DREAM Machine. If the neuron activation value is to be determined analytically, the performance will be further reduced.

The relatively similar execution rate obtained from the fixed ring mapping and the DREAM Machine mapping is due to the limited degree of parallelism available in the Nettalk network and its simple fully interconnected structure. This point becomes evident if we consider the percentage of optimality factor for both implementations. The optimal execution rate for implementing the Nettalk network is $T^*=263k_1+2k_2$ seconds, according to (5-13). This leads to an optimality factor of 100% for the DREAM Machine implementation vs. 52% for the fixed ring implementation.

5.10.3 Implementing a Block-Connected Multilayer Neural Network

A neural network model which reflects the current trend in structured network design has been proposed in [56] for image compression application. The structure of this network

(shown in Figure 5-8) is a good example of a blocked connected neural network used for data fusion, described in Section 5.6. This network consists of 5 layers. The input layer is comprised of 8 disjoint blocks of 64 neurons each. Each 64 neuron block in the input layer is fully connected to a unique 8 neuron block in the second layer. All the 64 neurons in the second layer are fully connected to n neurons in the third layer. The number of neurons in this layer (n) determines the amount of compression performed by the network. A symmetrically identical interconnection structure is constructed to decompress the information from the third through the fifth layers.

The ring structure for implementing this network on the DREAM Machine is shown in Figure 5-9. This ring structure operates in two configurations: Configuration 1 contains 8 non-intersecting rings of length 32 used for processing layers 2 and 5, thus $\tilde{R}_2 = \tilde{R}_5 = 32$. Configuration 2 is used for processing layers 3 and 4 and consist of a

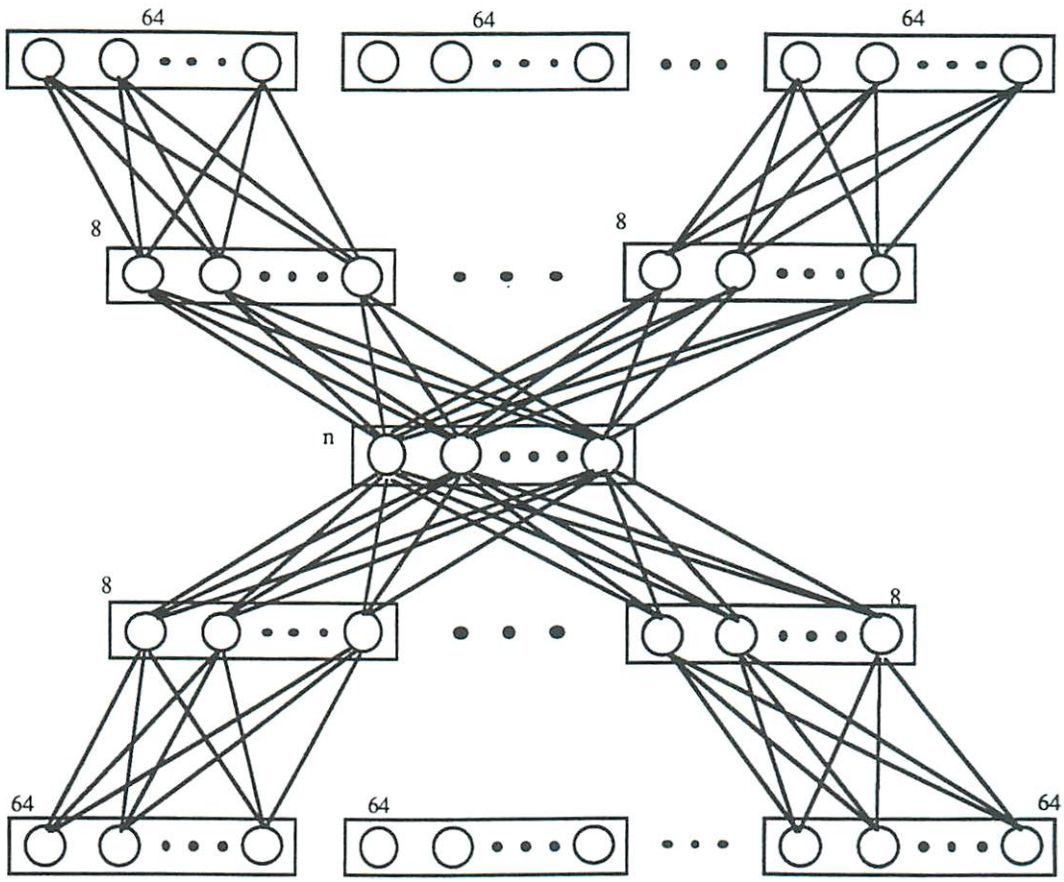


Figure 5-8 - A neural network structure for image compression and decompression with a regularly blocked structure [56].

single ring with 64 processors, thus $\tilde{R}_3 = \tilde{R}_4 = 64$. Virtual PEs must be used to implement the required 512 neurons in the input and output layers, since the processor array is assumed to have only 256 processors. The use of virtual PEs to implement layers 1 and 5 leads to $\tilde{\omega}_2 = \tilde{\omega}_5 = 2$. No other layers require virtual processors and therefore $\tilde{\omega}_3 = \tilde{\omega}_4 = 1$ and $\tilde{v}_2 = \tilde{v}_3 = \tilde{v}_4 = \tilde{v}_5 = 1$. Using equations (5-6) and (5-10) we can calculate the execution time required for implementing this network to be $T=256k_1+4k_2$ corresponding to a 598 MCPS throughput, assuming the compression factor n=64. Using a fixed ring architecture, the expected execution time will be $T=1536k_1+4k_2$, yielding a throughput rate of only 105 MCPS.

We can calculate the optimal execution time for this network taking advantage of all the available neuron level parallelism. This leads to a $T^*=256k_1+4k_2$ value. We can notice that similar to the Nettalk mapping, the DREAM Machine implementation can achieve 100% level of optimality where the fixed ring approach yields a performance of 18% of optimum.

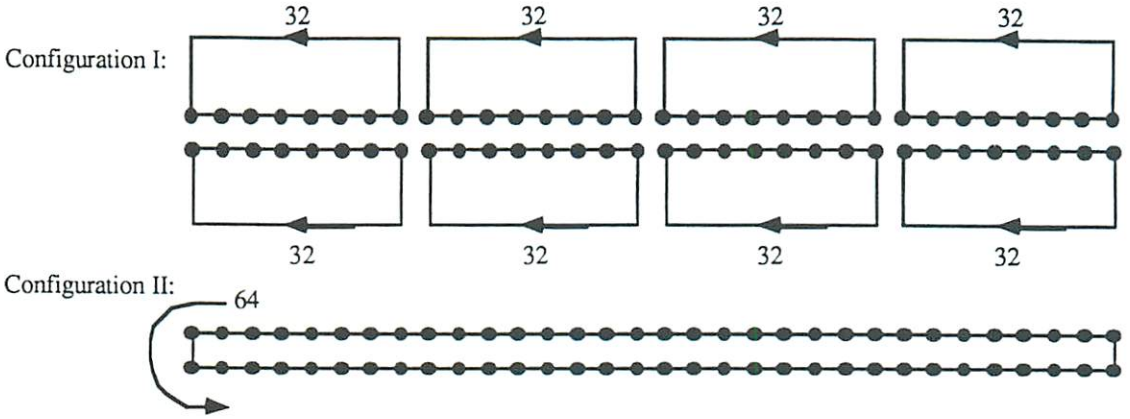


Figure 5-9 - The ring structure associated with the image compression and decompression network of Figure 5-8.

5.10.4 Implementing a Fully Connected Single Layer Network

The peak performance of the algorithmic mapping method on the DREAM Machine can be evaluated by mapping a neural network structure which offers the greatest amount of parallelism. One such network has been proposed in [33] where a fully connected neural

network is used to solve the Traveling Salesman Problem (TSP). The neural network requires N^2 neurons and thus N^4 synapses to implement an N city TSP. A 16 city TSP can efficiently utilize all the PEs of the DREAM Machine to arrive at a peak performance of 2,516 MCPS. Same type of performance can be expected from a fixed ring architecture when implementing this network since the ring size matches perfectly the number of available PEs in the machine.

To implement the 30 cities problem described in [33], we can construct a ring of size 225 (obtained using equation (5-4)) for processing the required 900 neurons. The amount of time required for each update of the neuron output values can be found via equation (5-3) to be $T=(4*4*225k_1 + 4k_2)$ = 361.8μs or 2,239 MCPS performance. Implementing this network on a 256 element fixed ring architecture leads to a 1,969 MCPS performance. This illustrates well the capability of the DREAM Machine and the variable ring mapping strategy to maintain a performance close to the peak value for neural network structures with varying sizes.

The measure of optimality can also be evaluated for this example using equation (5-13). This leads to $T^*=900k_1+1k_2$ =90.45μs value which assumes having 900 processors. In order to adjust for the actual number of processors we can multiply T^* by a factor of 900/256 = 3.52. This leads to a new T^* value of 318μs. Now the optimality ratio for the variable length ring mapping can be determined to be 88% and for the fixed length ring is 77%.

5.11 Analysis of Results

In this chapter we described a mapping method for efficient implementation of neural network models with regularly structured interconnections. We described how various features of the DREAM Machine architecture can be used to efficiently implement specific requirements imposed by neural network processing. The mechanism used for construction of processing rings of arbitrary size (less than the processing array size) was shown to be useful for matching the size of the problem to the architecture. The use of embedded ring structures, described in Section 5.4 and demonstrated in Section 5.10, have outlined the effectiveness of this method in significantly reducing the restrictions on the size and shape of the interconnection structure of multilayer neural networks for

efficient and fast processing. A summery of the performance comparison based on throughput rates is shown in Table 5-1 and comparison based on the optimality ratio is given in Table 5-2.

	Nettalk [69]	Compression Net [56]	Hopfield 30 Cites TSP [33]
Fixed-Size Ring Mapping	267 MCPS	105 MCPS	1,969 MCPS
Variable-Size Ring Mapping	512 MCPS	598 MCPS	2,239 MCPS

Table 5-1 - Performance comparison of the variable-size ring mapping vs. the fixed-size ring mapping based on the MCPS metric.

The performance of the DREAM Machine architecture was compared to the ring-connected systolic architecture in this chapter. Since both architectures use nearest neighbor communications, the extra communication hardware required by the DREAM Machine is only a constant factor greater than the 1-D ring architecture. On the other hand, the execution rate of the fixed ring was shown to be $O(P)$ where P is the number of processors in the network (assuming the number of neurons per layer is always less than P). Using the variable ring mapping method on the DREAM Machine, the execution rate of this implementation is $O(\tilde{R}^b)$, where $\tilde{R}^b$ is the size of the largest block in the network. As neural networks continue to be applied to more demanding applications requiring specialized structures for processing various portions of the input data, the number of blocks in a specific neural network will increase but the size of each block should remain relatively small and constant. In order to exploit all the available parallelism in such networks, the number of processors in the system should increase. The DREAM Machine architecture and the variable ring size mapping method support these demands very efficiently since the execution rate scales relative to the small and constant $\tilde{R}^b$ value rather than the large and growing processor array size.

	Nettalk [69]	Compression Net [56]	Hopfield 30 Cites TSP [33]
Fixed-Size Ring Mapping	52%	18%	77%
Variable-Size Ring Mapping	100%	100%	88%

Table 5-2 - Performance comparison of the variable-size ring mapping vs. the fixed-size ring mapping based on the optimality ratio.

Chapter 6

Optimization Based Mapping Method

Note: the work reported in this chapter and the associated results presented in Chapter 7 have been performed in collaboration with Dr. Petar Simic [74, 76]. In this chapter, we present a general method for mapping parallel algorithms onto parallel processing architectures. This mapping is performed in such a fashion as to optimize a specific objective function. The basic framework of this method is presented in a general form in order to be applicable to a wide range of problems in parallel processing. However, in this dissertation we specifically demonstrate the use of this method for implementing neural network models on the DREAM Machine architecture. The method is based on using neural computation techniques to arrive at good solutions to the mapping problem. The basic principles of this method and the specific cost functions used to measure the optimality of a specific mapping is described in detail. Simulation results of the algorithm are presented in Chapter 7.

6.1 Problem Overview

In general, parallel algorithms with specific regularities in their computation and interconnection patterns can be mapped onto parallel architectures manually by exploiting the inherent regularities of the algorithm. This type of an approach has been successfully used to arrive at efficient methods for implementing regularly structured algorithms, such as the Fast Fourier Transform (FFT) and the matrix-matrix-product operations, onto systolic parallel architectures [40]. In Chapter 5, we presented one such method for mapping neural network computation onto the DREAM Machine architecture. This

method utilized the regularity of the neural network interconnection structure to construct ring-shaped paths specifying the assignments of neurons to PEs and generating a simple flow pattern for transferring partial results between various processors.

6.1.1 The Mapping Problem

In mapping parallel algorithms, where no particular regularity in the interconnection patterns is apparent, two distinct approaches can be taken. One method is to enforce a certain regularity in the interconnection structure by introducing phantom processes and connections which do not contribute to the actual computation but are used strictly for correctly synchronizing the operations being performed by various processors. The analogy of this method for neural network processing is seen by the use of phantom neurons and synapses with zero output values and zero connection weights mentioned in Chapters 4 and 5. This method simplifies the mapping problem since this artificially created regularity can be used to ensure proper computation and still maintain the basic straight forward mapping algorithm. On the other hand, the introduction of phantom connections and processes to the computation, reduces the system efficiency by dissipating computational cycles on useless operations, such as multiplication and addition by zero.

The second method of addressing the problem of mapping irregularly structured algorithms is to abandon the use of the regularity in the interconnection structure all together. The mapping problem can then be stated generally as a problem of assigning various processes in a given algorithm to the various processors of a parallel machine and construction of conflict-free computational paths in such a fashion as to optimize a specific objective, such as having minimum execution time.

6.1.2 Complexity of the Mapping Problem

In general, the number of possible assignments of processes to processors, and the corresponding communication flow patterns is combinatorially high. For example, given a neural network with N neurons and a parallel architecture with P processors there are $P!/(P-N)!$ many different assignments that can be used to assign the neurons to the processors, assuming $P>N$ and each processor can hold no more than one neuron. The

space of possible configurations would be much larger if we allow for multiple neurons per processor configurations. Generally, there are N computational paths associated with neural network computation (one per neuron). There exist $P!N!/(P-M)!$ different configurations that can be used to construct these paths traversing various PEs in M communication steps. Of course, most of these assignments and flow patterns lead to very low system utilization and throughput rates. Only a handful of possible assignments and communication schedules lead to efficient and high throughput implementations.

6.1.3 Analogies to Other Combinatorial Optimization Problems

The problem of optimally mapping parallel processes onto parallel processing architectures is analogous to many other combinatorial optimization problems. The problem of assigning processes of a process graph to processors of an architecture, defined by an interconnection topology graph, is similar in nature to the sub-graph isomorphism problem. The relation between the assignment problem and sparse matrix bandwidth reduction has been illustrated in [7]. It has been shown that all of these problems fall into the class of NP-Complete problems where no polynomial time algorithm has been devised which can guarantee a solution to an arbitrary size problem.

As discussed in Section 6.1.2, the scheduling problem for implementing commutative computation according to the mapping method of Chapter 4, is of even larger complexity than the assignment problem. The construction of each computational path over the processing array can be viewed as a slight variant to the Traveling Salesman Problem (TSP), which is also known to be an NP-Complete problem [44]. Having to construct multiple paths while minimizing the longest path length is analogous to running multiple interdependent traveling salesman problems concurrently while optimizing the longest tour length.

Solving such combinatorially complex optimization problems has generally involve devising a heuristic and iterative approach which attempts at finding a "good" but not necessarily optimal solution [4, 7]. Starting from a random solution, these methods sequentially and slowly vary the system configuration so that at the completion of the algorithm no better solution is found. Due to the serial nature of these algorithms, their usage has been limited to small problems. In this chapter, we present a highly parallel method for efficiently searching the complex space of mapping possibilities in order to

find good solutions to the assignment and scheduling problems [78]. Similar but less general approaches have been proposed previously using various neural computation techniques [15, 33, 58].

6.2 General Approach

We have presented the general mapping approach used for implementation of neural networks on systolic parallel architectures in Chapter 4. This mapping method describes the general concept of the computation process by specifying the computational operations to be performed at each processor and the associated interprocessor communication requirements. However, this general mapping method does not specify the exact assignment of neurons to processors and the subsequent scheduling for the flow of data between the various processors. We now describe how we attempt to address each of these problems.

6.2.1 Mapping Parallel Algorithms Onto Parallel Architectures

The implementation of parallel algorithms on parallel processing architectures involves solving two different but interrelated problems. First, we need to determine what part of the total computation, associated with a specific algorithm, is to be performed by each of the processors in the target machine. Second, we need to establish how and in what order would these processors communicate with one another in order to implement the desired algorithm. In general, this problem can be broken down into three separate but again interrelated problems. The first one is sometimes referred to as the clustering problem. Given a specific algorithm as a collection of intercommunicating processes, the clustering problem involves grouping a number of processes into individual clusters such that the inter-cluster communications are minimized while computational parallelism is maximized. Of course, there is a trade-off between these two objectives, and the best trade-off depends on the specific characteristics of the target architecture.

With a specific clustering of processes into clusters, the second problem one must deal with involves finding an optimal assignment of clusters to processors of the target architecture such that the logical communications between clusters can be realized

efficiently by the physical interconnection topology of the architecture. This problem is referred to as the assignment problem [8]. The third and final problem, called the scheduling problem, involves specifying the order in which each process is executed and how the interprocessor communications are synchronized. The clustering and assignment problems must be solved before the scheduling task can begin. The objective of the scheduling problem is to order the computation and communication operations in such a manner as to have all the necessary computation completed in the shortest amount of time while observing specific physical limitations of the architecture, such as conflict-free communications.

Although there is a sequential order in which each of these problems must be solved, clustering, assignment, scheduling, all of these problems are interdependent on each other, and thus, the best implementation might require concurrent optimization of all three tasks. In the following sections we describe our approach for addressing each problem separately. However, our method is formulated such that concurrently solving for the best solution to all problems can also be performed. Below, we describe how the clustering problem can be addressed. The assignment and scheduling problems are treated separately in Sections 6.3 and 6.4, respectively.

6.2.2 Addressing the Clustering Problem

The need for solving the clustering problem arises in two different circumstances. First, if the number of processes of a given algorithm is larger than the number of processors in the target machine architecture, a method for clustering multiple processes into a single cluster is required. The second reason for performing the clustering operation is due to the specific architectural characteristics of the target machine. If the target machine's architecture is rather inhomogeneous, having computation and communication properties varying across the machine, clustering can be performed to group similar processes requiring similar architectural characteristics together. For example, with a machine architecture comprised of a group of analog processors and a group of digital processors, processes requiring high precision computation might be grouped together and assigned to digital processors. Other processes requiring certain characteristics which are better matched to analog hardware can be clustered and assigned to the analog processors.

The clustering problem has the same computational complexity as the assignment problem described in Section 6.1.2. Therefore, solving this problem for the optimal solution involves searching a combinatorially large space of possibilities. In this dissertation we do not attempt to solve the clustering problem for the general case of grouping neurons into clusters before performing the assignment procedure. As mentioned above, the clustering problem arises in cases of inhomogeneous algorithms and architectures, and in cases where the number of processes are larger than processors. For implementation of neural networks on systolic SIMD parallel architectures, both the algorithm and the architecture are rather homogeneous, thus eliminating the need for clustering due to the inhomogeneity.

For the case where the number of neurons exceed the number of processors, we use a trivial clustering method involving the generation of virtual processors via time multiplexing. With a machine having P processors, virtual processor P_v is processed at the $\lceil P_v / P \rceil + 1$ time slot of processor $(P_v \bmod P)$. Although this method does not guarantee any goodness of clustering, it is simple to implement. Optimization routines similar to those described for the assignment and scheduling problems later in this chapter can also be developed to address the clustering problem.

6.3 Solving the Assignment Problem

In this section we present a general formalism for solving the assignment problem. In general, any computation can be represented as a collection of processes performing specific operations on their local data items while communicating an arbitrary amount of information over a set of logical communication links. Similarly, a parallel processing architecture can be viewed as a collection of processing elements, operating and manipulating data items, being capable of communicating with one another over a physical interconnection network. The objective of the assignment procedure is to generate a mapping which assigns each process of the algorithm to a particular processor in the parallel architecture, such that a given criterion function is optimized. One such criterion function is to maximize the number of logical communication links being mapped to physical communication channels.

6.3.1 Problem Representation

A computational algorithm can be represent as a process graph $\boldsymbol{D}$ with nodes n,m=0, 1, 2, ··, N-1 representing individual process. These nodes are connected together via links D^{nm} denoting the logical data dependence between various processes, where $D^{nm}=1$ denotes a dependence between process n and process m. Similarly, the interconnection topology of a parallel processing architecture can be represented as a graph $\mathbf{G}$ with nodes A,B=0, 1, 2, ··, P-1 denoting the various PEs of the architecture, and links G^{AB} representing the physical communication channel between the processors. Any arbitrary interconnection topology can be specified by setting $G^{AB}=1$ to show the presence of a physical connection between processors A and B. With such a representation scheme, the assignment of process nodes to processor nodes can be represented by the use of a assignment matrix $\boldsymbol{\omega}$, having binary valued elements ω_n^A representing the assignment of process n to processor A, if and only if $\omega_n^A=1$. A graphical representation of this approach is shown in Figure 6-1.

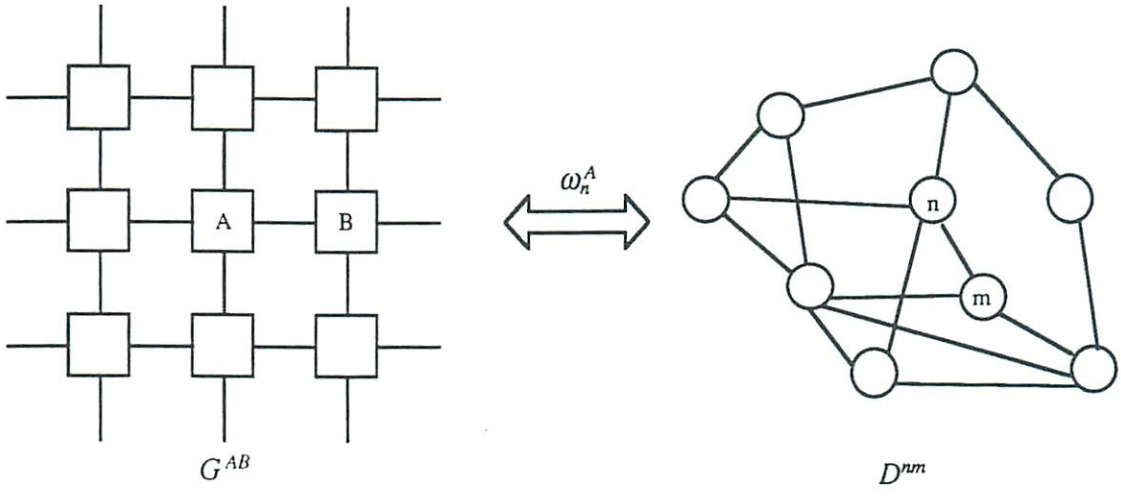


Figure 6-1 - Mapping of the nodes of a process graph **D** onto a processor graph **G** using the assignment matrix $\boldsymbol{\omega}$.

Let us further define a distance matrix $\mathbb{G}$ with elements $\mathbb{G}^{AB}$ representing the communication cost associated with transferring data from processor A to processor B. The distance measure can be arbitrarily specified according to the specific architectural features of the target architecture. For example, a measure which is used in our implementation is the city blocks distance between the various PEs. Other distance

measures, based on specific capacity of the various communication channels used by an architecture, can also be represented through the use of this matrix.

6.3.2 Assignment Cost Function Formulation

With the representation scheme described in the previous section, we can now proceed to design a specific cost function to assign the various nodes of the process graph to the nodes of the processor graph in such a way as to minimize the communication cost between the PEs. The ideal assignment would assign each process to a particular processor such that each of the logical connections in the process graph D^{nm} can be represented by a physical communication channel G^{AB} of the architecture. In general, we cannot guarantee that mappings which satisfy this condition do exist. For example, if a particular node of the process graph is connected to 5 other nodes of the graph, and the target computer architecture uses a four nearest neighbor interconnection topology, at best only four of the logical connections in the process graph can be directly mapped onto physical communication channels of the architecture. In such cases, we would like to assign the fifth node to a processor which is located at the shortest possible distance to its ideal position, hence the need for the distance matrix $\mathbf{G}$.

Given the graphical representation of the computation and the target architecture through the use of the **D** and **G** matrices, respectively, we can measure the amount of overlap between the links of the process graph and the links of the architecture graph given a particular assignment matrix $\boldsymbol{\omega}$. Specifically, we are interested in minimizing the number of mismatches between these graphs with a particular assignment. The extend of mismatch can be evaluated as

$$\sum_{A,B}\sum_{n,m}\left(G^{AB}-\omega_n^A D^{nm}\omega_m^B\right)^2, \tag{6-1}$$

where $\omega_n^A D^{nm}\omega_m^B = 1$ when process n is mapped onto processor A, process m is mapped onto processor B, and there exists a link between processes n and m. Thus, a matching in the assignment of logical process graph edges to physical processor graph edges does not contribute to this sum. As mentioned earlier, in cases where the links associated with each of the two graphs cannot be completely matched, we would like to map processes to processors physically close to the optimal location. This can be accomplished by

weighting the cost of each mismatched link by the distance between the two selected PEs as

$$\sum_{A,B}\sum_{n,m}\mathbb{G}^{AB}\left(G^{AB}-\omega_n^A D^{nm}\omega_m^B\right)^2. \tag{6-2}$$

In general $D^{nm}\neq D^{mn}$, corresponding to a directed process graph. However, we do require that the matrix **D** be quadratic of size NxN. By utilizing the binary nature of the **G** and **D** matrices, equation (6-2) can be simplified to

$$\sum_{A,B}\sum_{n,m}\left(-G^{AB}\mathbb{G}^{AB}\mathbb{D}^{nm}\omega_n^A\omega_m^B+\left(1-G^{AB}\right)\mathbb{G}^{AB}\mathbb{D}^{nm}\omega_n^A\omega_m^B\right), \tag{6-3}$$

where $\mathbb{D}^{nm}$ is defined as

$$\mathbb{D}^{nm}=\frac{1}{2}\left(D^{nm}+D^{mn}\right). \tag{6-4}$$

In this form G^{AB} is used to add or subtract the quantity $\mathbb{G}^{AB}\mathbb{D}^{nm}\omega_n^A\omega_m^B$ from the total cost, depending on its state. This can be simply implemented using a conditional statement requiring minimal amount of computation. In order to guarantee correct computation of this cost function, we require that both the distance matrix $\mathbb{G}$ and the process graph **D** be traceless, having elements $\mathbb{G}^{AA}=0$ and $D^{nn}=0$.

6.3.3 Constraints on the Assignment Matrix

There are a number of constraints placed on the structure of the assignment matrix which are required to ensure physically realizable solutions. In our formulation, we assume that the number of processors in the system is always greater or equal to the number of individual processes being implemented, $P\geq N$. We have already described how time multiplexing techniques can be used to increase the number of virtual PEs to the desired amount in order to satisfy this condition. A major constraint on the assignment matrix is to have each process be assigned to "some" processor in the architecture. This constraint can be written as

$$\sum_{A}\omega_n^A=1. \tag{6-5}$$

Later in section 6.6.1 we show how this condition can be strongly enforced in arriving at solutions to the assignment problem.

A second constraint is introduced by requiring that each processor not be assigned more than one process. This constraint can softly be enforced by having a penalty term

$$\sum_{n,m} d_{nm}\omega_n^A\omega_m^A \tag{6-6}$$

added to the cost function. In equation (6-6), the parameter d_{nm} is used to adjust the weight of this penalty term to the total cost of the assignment. Additional penalty terms can be added to ensure the compliance with other constraints. These include a term for discouraging mappings where one process is assigned to more than one processor by adding

$$\sum_{A,B} \gamma_{AB}\omega_n^A\omega_n^B \tag{6-7}$$

to the total cost. In both equations (6-6) and (6-7) we always assume that $\gamma_{AA} = 0$ and $d_{nm} = 0$. This ensures proper computation of the penalty terms.

If we assume that the number of processors in the network is equal to the number of processes in the process graph, then an additional penalty term of the form

$$\gamma_\omega \sum_A \left(1 - \sum_n \omega_n^A\right)^2 \tag{6-8}$$

can be added to the cost function. This term requires that each processor be assigned a single process. The parameters γ_ω and γ_{AB} are used to adjust the contribution weight of each term to the total cost. Equation (6-8) is only useful for the case where $P=N$. Otherwise, if $P>N$, then the cost associated with this term will never reach zero since some processes are not assigned to any processors.

In order to formulate equation (6-8) in a general form so that it is applicable to the more general case where $P \geq N$, we introduce a new binary valued variable η_A which indicates which of the P processors in the system are assigned the N processes. By selectively including only those PEs that are assigned a process, we can replacing equation (6-8) by

$$\gamma_\omega \sum_A \eta_A \left(1 - \sum_n \omega_n^A\right)^2. \tag{6-9}$$

We add an additional penalty term to the total cost function as

$$\gamma_\eta \left(N - \sum_A \eta_A\right)^2 \tag{6-10}$$

in order to ensure that only N processes are selected to participate in the calculation of equation (6-9).

In both equations (6-9) and (6-10), the diagonal elements of the quadratic operations, $\omega_n^A \omega_n^A$ and $\eta_A \eta_A$ respectively, must be removed from the computation in order to ensure proper convergence of the algorithm. This is accomplished by expanding each equation and individually subtracting these diagonal terms. The resultant form of equations (6-9) and (6-10) can be written as

$$\gamma_\omega \sum_A \eta_A \left(1 - \sum_n \omega_n^A\right)^2 + \gamma_\omega \sum_{A,n} \eta_A \omega_n^A \left(1 - \omega_n^A\right), \quad \text{and} \tag{6-11}$$

$$\gamma_\eta \left(N - \sum_A \eta_A\right)^2 + \gamma_\eta \sum_A \eta_A (1 - \eta_A), \tag{6-12}$$

respectively.

6.3.4 The Complete Assignment Cost Function

The complete cost function used as the criterion of our assignment procedure can be assembled by combining the penalty terms given in Section 6.3.3 with the weighted mismatch cost given by equation (6-2). The final form of this cost function with appropriate weighting constants is given by

$$\begin{aligned} C_\omega(\boldsymbol{\omega}) = & \frac{1}{4} \sum_{A,B=0}^{P-1} \mathcal{G}^{AB} \left(G^{AB} - \sum_{n,m=0}^{N-1} \omega_n^A D^{nm} \omega_m^B\right)^2 \\ & + \frac{1}{2} \sum_{n,m=0}^{N-1} d_{nm} \omega_n^A \omega_m^A + \frac{1}{2} \sum_{A,B=0}^{P-1} \gamma_{AB} \omega_n^A \omega_n^B \\ & + \frac{1}{2} \gamma_\omega \sum_{A=0}^{P-1} \eta_A \left(1 - \sum_{n=0}^{N-1} \omega_n^A\right)^2 + \frac{1}{2} \gamma_\omega \sum_{A=0}^{P-1} \left(\eta_A \sum_{n=0}^{N-1} \omega_n^A \left(1 - \omega_n^A\right)\right) \\ & + \frac{1}{2} \gamma_\eta \left(N - \sum_{A=0}^{P-1} \eta_A\right)^2 + \frac{1}{2} \gamma_\eta \sum_{A=0}^{P-1} \eta_A (1 - \eta_A) \end{aligned} \tag{6-13}$$

This formulation of the assignment cost function can be viewed as a method for solving the sub-graph isomorphism problem, with the additional twist of having a weight factor for the mismatched nodes having a magnitude proportional to the distance between the

ideal and the actual assignment. Other methods based on solving the sub-graph isomorphism problem for assigning parallel processes onto parallel architectures have been previously explored using heuristic techniques, such as [7]. A comparison of our method with the method of [7] is given in Chapter 7.

The basic function performed by this assignment cost function is to assign the processes that are connected to each other, defined by matrix $\mathbf{D}$, to a cluster of processors located in a small neighborhood of each other. For example, let us assume that process n is assigned to processor A, $\omega_n^A = 1$. Let $S_n = \{m | D^{nm} \neq 0\}$ represent the set of all processes that have a logical connection to process n, and define K_n to be the number of elements in the set S_n. If each of the K_n elements of set S_n is assigned to processors in the local neighborhood of processor A, then the cost associated with this mapping is zero, according to the first term in equation (6-13). Otherwise, each of the processes mapped to a non-neighboring processor of processor A, call this processor C, increases the assignment cost by a value proportional to the distance between processor A and C, given as $\mathcal{G}^{AC}$.

There are several cases where such non-optimal mappings occur. The most obvious one is when K_n is greater than the number of physical connections associated with each processor of the architecture. The second case is related to the complexity of the process graph. If a single process is logically connected to a large number of otherwise disjoint process clusters, the competition between these clusters will cause the particular process to be assigned to a *compromise* processor that will keep the total cost to a minimum. Of course, there are cases where no optimal assignment can be achieved due to the intrinsic differences between the process and processor graphs. In such cases, it is impossible to find a mapping which completely satisfies the first term in equation (6-13). An example of such a case is given in section 7.2.

6.3.5 Assignment of Neurons to Processors

The assignment cost function, described in the preceding sections, can be utilized to find good mapping solutions for assigning neurons of a neural network to processors of a parallel processor. This can be accomplished by treating each neuron as a single process of the process graph and each synaptic weight in the $\mathbf{W}$ matrix as an edge of this graph. In this fashion, each non-zero synaptic connection between neurons n and m is

represented by a D^{nm} element set to one. This representation scheme can be directly applied to single layer neural network models, such as the Hopfield net [33]. Application of this approach to layer structured neural networks requires a relabeling of neurons as processing propagates through the various layers of the network.

The formulation of the assignment cost function as presented in equation (6-13) attempts to assign the neurons $m \in S_n$ to processors in the local neighborhood of neuron n. In mapping schemes where layer level parallelism is not utilized, it is inefficient to have neurons of different layers mapped to different processors since processing associated with each different layer is performed in series. Therefore, neuron n of layer ℓ which is connected to neurons $m \in S_n$ of layer ℓ-1 is assigned the same processor as neuron n of layer ℓ-1. This restriction is not significant for iterative neural network models, such as the BAM [38] and ART [10] models, but can cause inefficiency in layered networks with large differences between the number of neurons in various layers. However, this inefficiency is primarily due to the computation associated with the implementation of the assignment procedure and not the actual assignment solution. In any case, it is possible to introduce yet another variable to the cost function to automatically determine the best assignment of neurons in layered structures.

6.4 Solving the Scheduling Problem

As mentioned previously in Section 6.1, the scheduling problem involves determining an optimal sequence of operation and data movements among processors, such that a specific criterion is optimized. The most common criteria used for the scheduling problem is that of the execution time, which is to be minimized. More complex criterion functions, based on specific needs and architectural characteristics of a particular problem, can also be devised using a similar approach to the one described here.

6.4.1 General Approach

Given a particular assignment of processes to processors of a parallel architecture, we are interested in generating a scheduling of operations and flow of data between the processors such that the total execution time of the algorithm is minimized. A data

dependence diagram (or a data-flow graph) is required in order to generate a computation and communication schedule. Figure 6-3 depicts a typical data dependence graph for a simple calculation.

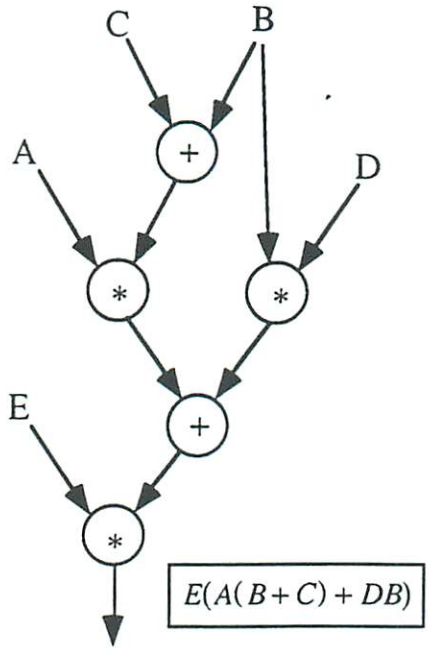


Figure 6-3 - An example data dependence graph for a simple arithmetic calculation

A general approach for evaluating a cost associated with the time necessary to traverse a data dependence graph can be derived similar to the one used for solving the assignment problem. The goal of such a cost function would be to schedule as many operations as possible such that they perform their operations in parallel. A more complicated scheduling problem is encountered when implementing associative or commutative operations, such as multiplication and addition operations, in a distributed fashion. In such cases, the order in which each particular summation or multiplication operation is perform is irrelevant and therefore the number of possible schedules for this computation is combinatorially large. Figure 6-4 shows multiple dependence graphs that accomplish the same commutative computation.

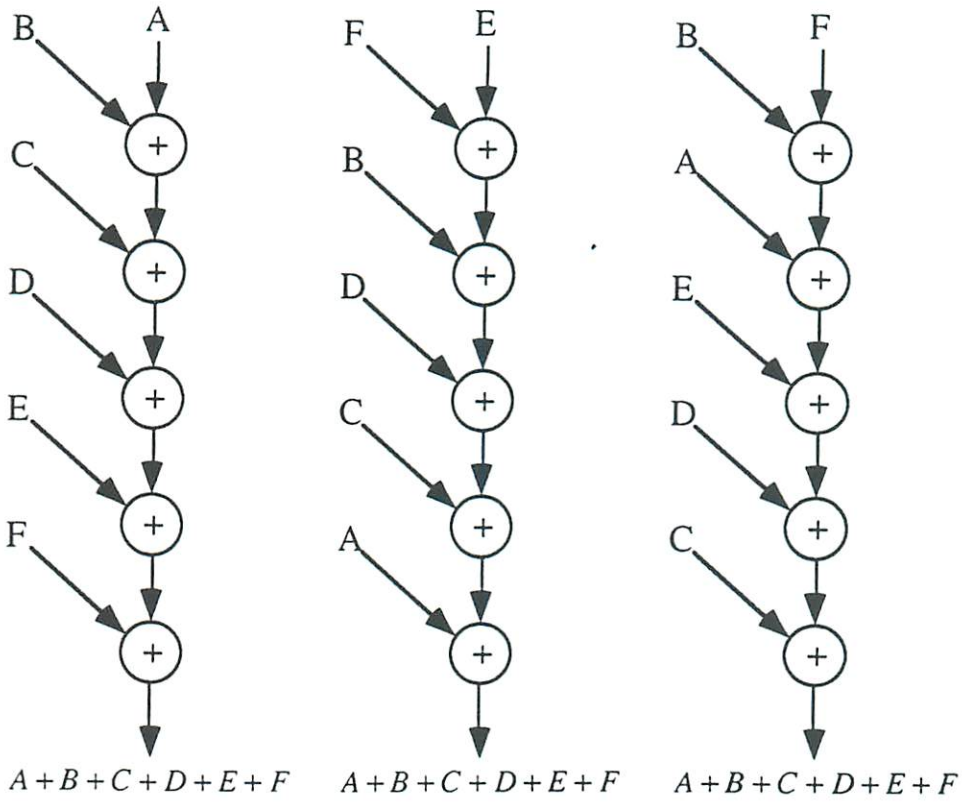


Figure 6-4 - Several different dependence graphs performing the same computation due to the commutative nature of the addition operation.

In this dissertation, we limit our discussion to the details of the scheduling problem for this special case of commutative operations. This is primarily due to the fact that neural computation involves the calculation of the weighted input sum of a neuron and the mapping method we have selected for parallel implementation of neural networks, described in Section 4.1, takes advantage of the commutative nature of the addition operation.

6.4.2 Specific Problem Representation

In Section 4.1, we presented the general mapping method used for implementing neural networks on systolic parallel architectures. The scheduling problem for implementing commutative operations on a systolic parallel architecture, where the results of the computation is not dependent on the specific order in which each of the operations is executed, involves determining the specific sequence in which the computation flows

from one processor to the next, regardless of the order in which each processor is traversed. In addition to neural network computation, this approach can also be applied to other similar algorithms, such as solving linear systems of equations [74].

There are a certain number of specific assumptions made in our formulation of the scheduling problem. First, the processes mapped to processors are assumed to require only a unit time in order to perform their computation. Second, each processor can autonomously select which of its neighboring processors to communicate with during each communication cycle. Finally, each processor can only perform operations associated with a single computation during each processing cycle. Given these assumptions and the mapping principle presented in Section 4.1, the scheduling problem involves formation of computational paths, each path associated with the computation of the weighted input sum to a specific neuron. Paths are constructed such that no two paths cross the same processor at the same time, and the data movement is limited to be between processors physically connected to one another through a shared communication

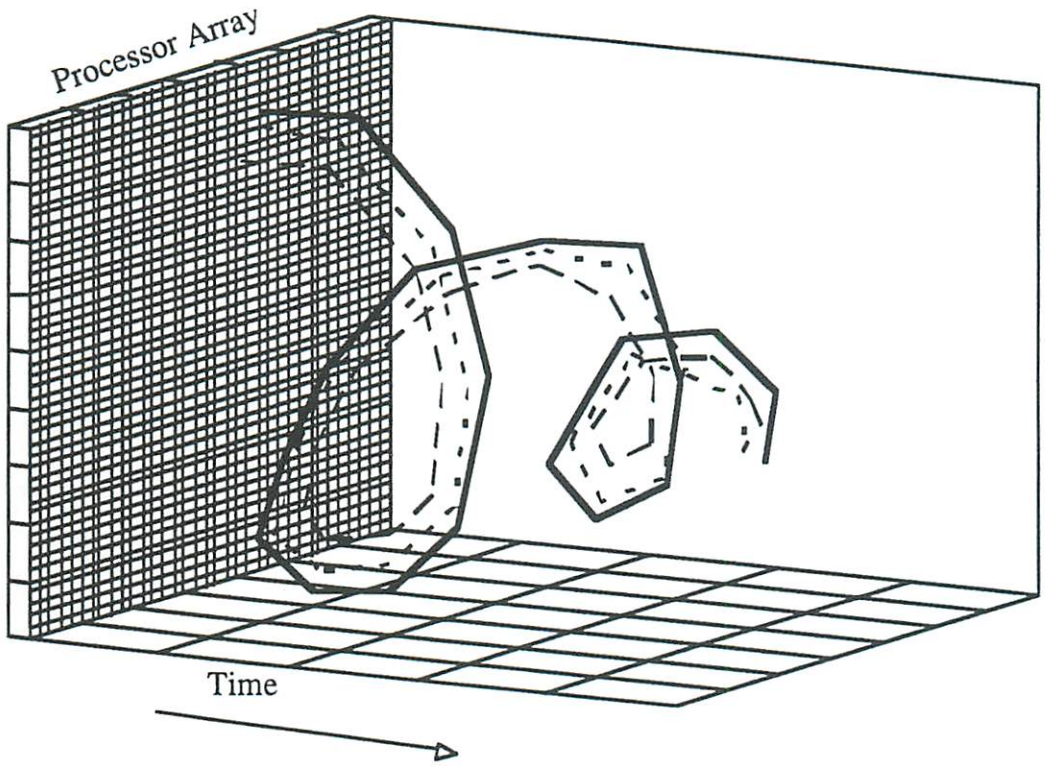


Figure 6-5 - A 3-D representation of 3 non-intersecting path traversals in time. The left hand plane represents the 2-D processor array

channel. The criteria used to optimally arrive at these computational paths is to schedule the flow pattern such that the length of the longest path is minimized while all the restriction of legal data movement between processors are adhered to.

Figure 6-5 graphically displays an example of 3 non-intersecting computational paths being traversed concurrently. The 2-D processing array is augmented by the time axis for better representation. The computational paths depicted in this figure follow a simple flow pattern where all the paths move in the same direction and are offset by a few processors in the space dimension.

6.4.3 Scheduling Cost Function Formulation

A similar approach to that of the assignment cost function formulation is used here to solve the scheduling problem. The symbol α=0, 1, 2, ··, A-1($\leq N$) is used to denote one of the A computational paths in the neural network. The **G** matrix is again used to specify the interprocessor communication topology of the target architecture of the mapping. The dependence graph used for the scheduling problem is specified via the **D** matrix. However, in the formulation of the scheduling cost function, each element of the **D** matrix $D^{\alpha n}$ represents the need for the computational path α to traverse neuron n. In other words, $D^{\alpha n} = 1$ denotes that neuron n participates in the computation associated with path α.

For implementing commutative operations in an iterative system, such as single layer feed-back neural network models, $D^{\alpha n}$ is equivalent to the D^{nm} matrix used for the assignment problem, since N=A. Although the case where A<N can also be implemented using the same procedure, by augmenting the matrix with zeros, in the remainder of this section we assume that the matrix **D** is quadratic of size NxN.

The objective of solving the scheduling problem is to arrive at a solution which specifies, for each path, the time order in which each processor is to be traversed. This can be represented by a 3-D matrix $\boldsymbol{\theta}$ with binary valued elements $\theta^{i}_{n(\alpha)}$. This variable has a value of one when at time i path α traverses neuron n. In order to avoid the use of non-linear functions in our calculation of the scheduling cost function, no direct method for measuring the length of each path, passing through all of its necessary processors, can be constructed. Therefore, we have selected to use an iterative approach to finding *good* solutions that traverse all the necessary processor traversals in a prespecified

number of time steps, M. The bounds on the value of M can be derived by considering that the lower bound on M corresponds to the number of neurons (or processors) to be traversed by the path α^*, where α^* denotes the path having the largest number K^* of neurons to be traversed. The upper bound on M is obviously equal to the maximum number of neurons in the network N. Therefore, we can set the value of M to be in the range $K^* \leq M \leq N$. The actual value of M depends strongly on the complexity of the neural network interconnection structure and the number of physical communication channels per processor. In practice, we can choose M to be fairly close to K^*, for example $1.2K^*$.

This formulation leads to a $\boldsymbol{\theta}$ matrix of size AxNxM. As stated earlier, assuming iterative algorithms, path α must be at the processor assigned neuron α. This can simply be implemented in our calculation by having

$$\theta_n^{M-1}(\alpha) = \delta_{an}, \qquad \text{where} \tag{6-14}$$

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{Otherwise} \end{cases}. \tag{6-15}$$

Equation (6-14) indicates that at the final time step (M-1), path α is in processor assigned neuron $n=\alpha$. This condition establishes the end point of each path, however, the initial point where path α begins ($\theta_n^0(\alpha)$) is not fixed and is left to be dynamically determined by the optimization algorithm.

According to our representation scheme, the scheduling algorithm logically specifies data movement from one neuron to the next. The specific assignment of neurons to processors must be considered in order to determine if there is a physical connection between to neurons by having been assigned to two neighboring processors. This can be accomplished through the use of the matrix

$$T_{nm}(\boldsymbol{\omega}) = \delta_{nm} + \sum_{A,B=0}^{P-1} \omega_n^A G^{AB} \omega_m^B, \quad \forall \; n,m\text{: } 0, 1, \cdots, P-1 \tag{6-16}$$

assuming that no processor is connected to itself ($G^{AA} = 0 \quad \forall \; A$: $0, 1, \cdots, P-1$).

In order to ensure proper calculation of the scheduling cost function, we require that the dependence graph be traceless having all diagonal elements set to zero ($D^{\alpha a} = 0 \; \forall \quad \alpha$: $0, 1, \cdots, A-1$). In case a neuron has a feedback connection to itself, the operation associated with this connection can be executed during the last cycle of

processing and be treated as a local operation. This approach is possible since each path must end in the processor holding its neuron output value according to equation (6- 14).

A particular path α must pass through all the processors holding neuron output values of neurons specified by $D^{\alpha n}$ in M-1 time steps. We can define a cost function which measures the number of times path α passes through processors that are not necessary for the computation of neuron α. This is given by

$$\sum_{i=0}^{M-2}\sum_{n,m=0}^{P-1} T_{nm}(\boldsymbol{\omega})\left(1-D^{\alpha m}\right)\theta^{i}_{n(\alpha)}\theta^{i+1}_{m(\alpha)} \tag{6-17}$$

The minima of this cost function for a specific path α represents a path which traverses all the necessary neurons required for its evaluation. Equation (6-17) counts only does jumps between neurons being connected via a physical communication channel, specified by $T_{nm}(\boldsymbol{\omega})$. Each time path α moves from neuron n at time i to neuron m at time i+1, a constant value of one is added to the cost, if neuron m is not necessary for the computation of path α. Otherwise, if neuron m is required for the computation of path α, no cost is added for going to that neuron.

Equation (6-17), however, is not concerned with where a particular path starts its traversal. We would like to introduce a term to the cost function so that paths starting in a processor which is assigned a neuron contributing to the path's computation would have a lower cost value. This can simply be done by adding the term

$$\lambda_0 \sum_{n=0}^{P-1}\left(1-D^{\alpha m}\right)\theta^{0}_{m(\alpha)} \tag{6-18}$$

to the cost function. Here the parameter λ_0 is used to adjust the weight associated with this term to the total cost of the scheduling function.

As mentioned in the previous section, equation (6-17) measures the number of times path α passes through unnecessary processors. As it stands, this cost function has the same cost for a scheduling which produces a path that remains in one of the processors required for its computation during the entire M cycles, or a path which traverses all of the K_α neurons required by path α. Clearly, the computation associated with path α requiring neuron n can be performed during the first cycle when path α crosses the processor assigned neuron n. All other instances where path α crosses the processor of neuron n, should be considered equivalent to passing through a processor which is not required by path α. In order to accomplish this, we augment the cost function with a

penalty term counting the number of times path α crosses the same neuron. This is given by

$$C_x = \sum_{i=0}^{M-2}\sum_{m=0}^{P-1}\left(D^{\alpha m}\theta_m^{i+1}(\alpha)\sum_{\ell=0}^{i}\theta_m^{\ell}(\alpha)\right). \tag{6-19}$$

We can notice that equation (6-19) contributes zero to the cost the first time path α crosses neuron m. During the subsequent crossings of neuron m by path α, equation (6-19) adds a penalty equivalent to the number of additional crossings to the total cost. In this fashion, each extra crossing of path α through neuron m contributes a larger value to the cost. This is not exactly the desired effect of having subsequent crossings be equivalent to passing through a non-contributing neuron. In order to address this problem we introduce a new term which subtracts the extra penalty occurred by more than two crossing of neuron m by path α. This term is formulated as

$$C_{\bar{x}} = \sum_{i=0}^{M-2}\sum_{m=0}^{P-1}\left(\eta_m^i(\alpha) D^{\alpha m}\theta_m^{i+1}(\alpha)\left(1-\sum_{\ell=0}^{i}\theta_m^{\ell}(\alpha)\right)\right). \tag{6-20}$$

In equation (6-20) we have introduced a new binary valued variable $\eta_m^i(\alpha)$ which has a value of one after the second crossing of neuron m by path α. Figure 6-6 shows the scheduling flow of path α which crosses a neuron m multiple times, assuming that neuron m is required for the computation of path α ($D^{\alpha m}=1$). It can be seen through this example that the total cost that would be added to the scheduling cost function will remain a constant value, with multiple crossings, when C_x and $C_{\bar{x}}$ values are added together.

In order to find good solutions to the scheduling problem, we would like to minimize the total cost associated with the terms described by equations (6-17) through (6-20). There are a number of constraints that must also be satisfied in order to correctly accomplish this task. In the following section we describe each of these constraints individually.

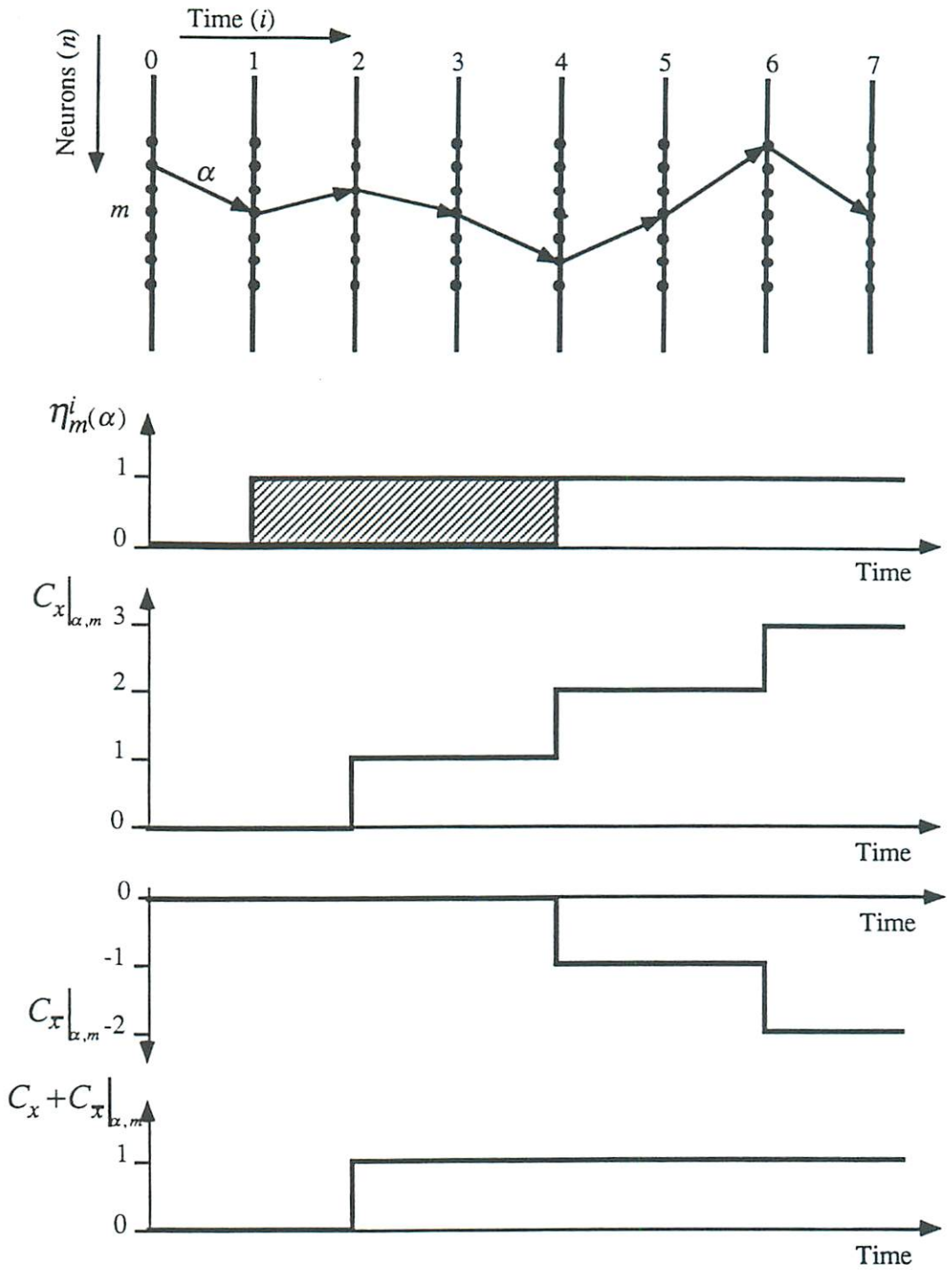


Figure 6-6 - An example of a path traversal and its associated penalty term contributions due to repeated crossings of a neuron m required for its computation. The value of $\eta^i_{m(\alpha)}$ can be any value in the range [0,1] in the shaded region.

6.4.4 Constraints Terms of the Scheduling Cost Function

The constraint terms associated with the optimization procedure, used for solving the scheduling problem, can be grouped into two categories. First, there are a number of constraints imposed by the physical realization of the scheduling routine, such as limiting the flow of data between processors that are physically connected to each other and allowing only a single process to be executed in each processor during a single time slot. The second type of constraints are used to enforce the legality of solutions produce by the optimization routine, such as not allowing a single path to be in two different processors at the same time. We will now give a detail description of these various constraints.

6.4.4.1 Architecture Dependent Constraints

It can be seen from Figure 6-6 that $\eta^i_{m(\alpha)}$ can take on any value during the time between the first instance that a path crosses a given neuron to just before the third time the path crosses the same neuron again. In the actual implementation of the optimization routine, $\eta^i_{m(\alpha)}$ is a continuous valued variable in the range [0,1]. In order to keep the value of $\eta^i_{m(\alpha)}$ to a zero or one value, we introduce the following penalty term:

$$\beta_\eta \sum_{i=0}^{M-2} \sum_{m=0}^{P-1} D^{\alpha m} \eta^i_{m(\alpha)} \left(1 - \eta^i_{m(\alpha)}\right), \tag{6-21}$$

where β_η is a scaling parameter.

In equation (6-17) we assumed that all data transfers between neuron n and neuron m occur when there is a physical communication channel between the two processors having been assigned the values of these neurons. In actuality, it is not possible to directly guarantee that this condition is always met. In fact, if path α moves from neuron n to neuron m, where there is no physical connection between these two neurons ($T_{nm}(\boldsymbol{\omega}) = 0$), no cost would be incurred by the cost function of equation (6-17). In order to ensure that paths only move between processors that are physically connected, we introduce a penalty term which adds a strong penalty to the total cost function whenever "illegal jumps" are made. This penalty term is similar to equation (6-17) and is formulated as

$$\lambda_\infty \sum_{i=0}^{M-2} \sum_{n,m=0}^{P-1} \left(1 - T_{nm}(\boldsymbol{\omega})\right) \theta^i_{n(\alpha)} \theta^{i+1}_{m(\alpha)}. \tag{6-22}$$

The parameter λ_{∞} is used to adjust the magnitude of the amount of contribution to the total cost due to this term. This value should generally be set relatively high, since scheduling solutions with illegal jumps are not physically realizable.

A similar constraint, which must be satisfied to ensure realizable solutions, is to allow only a single path to cross a given processor during each cycle. In other words, we would want to ensure that at each time step no two paths attempt to jump to the same neuron processor. This constraint can also be represented through a penalty term which contributes a large cost value to the scheduling cost if more than one path cross the same processor at the same time. The form of this penalty term is written as

$$\sum_{\alpha,\beta}^{A-1}\sum_{i=0}^{M-2}\sum_{n=0}^{P-1}\gamma^{\alpha\beta}\theta_n^i(\alpha)\theta_n^i(\beta), \tag{6-23}$$

where $\gamma^{\alpha\beta}$ is a scaling parameter having $\gamma^{\alpha\alpha}=0$. Similar to the λ_{∞} parameter in equation (2-22), the value of $\gamma^{\alpha\beta}$ should be high relative to the other terms in the cost function. Furthermore, we weight all path crossings equally, and therefore can define $\gamma^{\alpha\beta}\equiv\gamma_A\left(1-\delta^{\alpha\beta}\right)$.

6.4.4.2 Constraints on Path Variables

Earlier in this section we introduced the binary valued variable $\theta_n^i(\alpha)$ indicating the crossing of path α at neuron n at time i. In order to correctly calculate the desired cost function of equation (6-17), we must ensure that at each time step, each path is in one and only one processor, associated with some neuron n. This can be accomplished by having

$$\sum_{n=0}^{P-1}\theta_n^i(\alpha)=1. \tag{6-24}$$

We will show how this constraint can be directly built into our optimization procedure in Section 6.5. The solutions generated by this procedure will always guarantee the validity of equation (6-24).

Another constraints that is placed on the path variable $\theta_n^i(\alpha)$ is to not allow the same path α to be in two different locations at the same time i. This constraint can be enforced similar to equation (6-6) used for the assignment problem, by adding the term

$$\sum_{i=0}^{M-2}\sum_{m=0}^{P-1} d_{nm}^{\theta}\theta_n^i(\alpha)\theta_m^i(\alpha) \tag{6-25}$$

to the scheduling cost function. Here again, the parameter d_{nm}^{θ} is a scaling parameter with $d_{nn}^{\theta}=0$. A complementary constraint for ensuring that A processors are occupied with A paths during each cycle can be accomplished in the same manner used for solving a similar constraint of the assignment problem via equations (6-11) and (6-12). The corresponding equations for the scheduling problem are

$$\gamma_{\theta}\left(\sum_{i=0}^{M-2}\sum_{n=0}^{P-1}\varepsilon_n^i\left(1-\sum_{\alpha=0}^{A-1}\theta_n^i(\alpha)\right)^2+\sum_{i=0}^{M-2}\sum_{n=0}^{P-1}\varepsilon_n^i\theta_n^i(\alpha)\left(1-\theta_n^i(\alpha)\right)\right) \text{ and} \tag{6-26}$$

$$\gamma_{\varepsilon}\left(\sum_{i=0}^{M-2}\left(A-\sum_{n=0}^{P-1}\varepsilon_n^i\right)^2+\sum_{i=0}^{M-2}\sum_{n=0}^{P-1}\varepsilon_n^i\left(1-\varepsilon_n^i\right)\right), \tag{6-27}$$

where the last term in each equation is required to remove the diagonal elements of the $\boldsymbol{\theta}$ and $\boldsymbol{\varepsilon}$ matrices. In equations (6-26) and (6-27), γ_{θ} and γ_{ε} are parameters used to adjust the degree of the contribution made by each term to the total cost function.

The constraint associated with having each path passing through some neuron processor at each time step was given earlier through equation (6-24). As stated previously, our optimization procedure is design so that this condition is guaranteed at the completion of the procedure. We can also add a corresponding penalty term, of small magnitude, to the total cost function so that the restrictions imposed by this constraint are softly enforced during formation of a solution. This can be accomplished by adding the term

$$\gamma_{p}\left(\sum_{i=0}^{M-2}\left(1-\sum_{n=0}^{P-1}\theta_n^i(\alpha)\right)^2+\sum_{i=0}^{M-2}\sum_{n=0}^{P-1}\theta_n^i(\alpha)\left(1-\theta_n^i(\alpha)\right)\right) \tag{6-28}$$

to the total cost function, where the second term is again used to remove the extra diagonal elements from the computation.

6.4.5 The Complete Scheduling Cost Function

The cost function associated with scheduling the flow of concurrently flowing computational paths through a processing array, with a specific interconnection topology and a predefined assignment of processes to processors, can be assembled by putting

together the various terms described in the previous section. Given a specific assignment matrix $\boldsymbol{\omega}$ a good solution to the scheduling problem can be found by minimizing the cost function

$$
\begin{aligned}
C_\theta(\theta|\boldsymbol{\omega}) = \quad & \lambda_0 \sum_{n=0}^{P-1} \left(1 - D^{\alpha m}\right) \theta^0_{m(\alpha)} + \frac{1}{2} \sum_{i=0}^{M-2} \sum_{n,m=0}^{P-1} T_{nm}(\boldsymbol{\omega}) \left(1 - D^{\alpha m}\right) \theta^i_{n(\alpha)} \theta^{i+1}_{m(\alpha)} \\
+ \quad & \frac{1}{2} \sum_{i=0}^{M-2} \sum_{m=0}^{P-1} \left(D^{\alpha m} \theta^{i+1}_{m(\alpha)} \sum_{\ell=0}^{i} \theta^{\ell}_{m(\alpha)} \right) \\
+ \quad & \frac{1}{2} \sum_{i=0}^{M-2} \sum_{m=0}^{P-1} \left(\eta^i_{m(\alpha)} D^{\alpha m} \theta^{i+1}_{m(\alpha)} \left(1 - \sum_{\ell=0}^{i} \theta^{\ell}_{m(\alpha)} \right) \right) \\
+ \quad & \frac{1}{2} \beta_\eta \sum_{i=0}^{M-2} \sum_{m=0}^{P-1} D^{\alpha n} \eta^i_{m(\alpha)} \left(1 - \eta^i_{m(\alpha)}\right) \\
+ \quad & \frac{1}{2} \lambda_\infty \sum_{i=0}^{M-2} \sum_{n,m=0}^{P-1} \left(1 - T_{nm}(\boldsymbol{\omega})\right) \theta^i_{n(\alpha)} \theta^{i+1}_{m(\alpha)} \\
+ \quad & \frac{1}{2} \sum_{\alpha,\beta}^{A-1} \sum_{i=0}^{M-2} \sum_{n=0}^{P-1} \gamma^{\alpha\beta} \theta^i_{n(\alpha)} \theta^i_{n(\beta)} \\
+ \quad & \frac{1}{2} \sum_{i=0}^{M-2} \sum_{m=0}^{P-1} d^{\theta}_{nm} \theta^i_{n(\alpha)} \theta^i_{m(\alpha)} \\
+ \quad & \frac{1}{2} \gamma_\theta \left(\sum_{i=0}^{M-2} \sum_{n=0}^{P-1} \varepsilon^i_n \left(1 - \sum_{\alpha=0}^{A-1} \theta^i_{n(\alpha)} \right)^2 + \sum_{i=0}^{M-2} \sum_{n=0}^{P-1} \varepsilon^i_n \theta^i_{n(\alpha)} \left(1 - \theta^i_{n(\alpha)}\right) \right) \\
+ \quad & \frac{1}{2} \gamma_\varepsilon \left(\sum_{i=0}^{M-2} \left(A - \sum_{n=0}^{P-1} \varepsilon^i_n \right)^2 + \sum_{i=0}^{M-2} \sum_{n=0}^{P-1} \varepsilon^i_n \left(1 - \varepsilon^i_n\right) \right) \\
+ \quad & \frac{1}{2} \gamma_p \left(\sum_{i=0}^{M-2} \left(1 - \sum_{n=0}^{P-1} \theta^i_{n(\alpha)} \right)^2 + \sum_{i=0}^{M-2} \sum_{n=0}^{P-1} \theta^i_{n(\alpha)} \left(1 - \theta^i_{n(\alpha)}\right) \right)
\end{aligned}
\tag{6-29}
$$

with respect to the variables $\theta^i_{n(\alpha)}$, $\eta^i_{n(\alpha)}$, and ε^i_n. The complete scheduling cost function described by equation (6-29) involves a large number of terms and a significant amount of associated computations. In Chapter 8 we will describe the relative computational complexity of this function as it relates to solving difficult scheduling problems.

A good solution to the scheduling problem is obtained by finding a specific configuration of the variables $\theta^i_{n(\alpha)}$, $\eta^i_{n(\alpha)}$, and ε^i_n, such that the cost $C_\theta(\theta|\boldsymbol{\omega})$ is at a minimum. A good solution refers to a scheduling which ensures that all architectural

specific constraints and all the constraints on the path variables are meet and, at the same time, all paths are scheduled such that the maximum number of neurons associated with the computation of each path is traversed by that path in a specified amount of time (M). In Section 6.4.3, we discussed the bounds on the time limit parameter M. Finding optimal solutions to the scheduling problem in practice involves an iterative processes where we attempt to find good solutions in a specified time limit M, and dynamically varying the value of M between consecutive execution of the optimization procedure. The objective here is to find the smallest value of M for which the optimization procedure can generate legal solutions while all paths completely traverse their necessary neuron processors.

When implementing algorithms such as the sparse vector-matrix-product operation, if the discrepancy between the total number of elements in the vector (N) and the largest number of elements which participate in a particular path (K^*) is large $K^* << N$, then sufficient improvement in efficiency and throughput can be achieved even with M being several times the size of K^*.

6.4.5 Use of the Scheduling Procedure for Mapping Neural Networks

The scheduling cost function presented in the previous section can be used for arriving at data flow patterns on a systolic architecture for implementation of commutative calculations, where the solution of the computation is not dependent on the order in which its component operations are executed. In Chapter 4, we presented a mapping approach for implementing neural computation on systolic parallel architectures which required generation of computational paths similar to the ones described above. Since we have described the scheduling problem in terms of neural network computation, in this section we discuss the use of this cost function for implementing various neural network structures.

The formulation of the scheduling cost function can be directly applied to finding concurrently traversing paths on a distributed processing architecture following the mapping method of Chapter 4. In this formulation, distributed aspects of neural computation is viewed as being equivalent to performing a vector-matrix-product operation. This approach can directly be applied to single layer neural network models with feed-back processing, such as the Hopfield network [33]. Due to the iterative

nature of the computation of these models, the computational path α associated with a neuron n must end in the processor holding the output value of neuron n. This is exactly what is implemented by enforcing the constraint of equation (6-14).

In implementing feed-forward neural networks, with a single layer of synaptic interconnections, such as the Perceptron network [65], there is no specific requirement on the paths specifying the processor in which each path terminates. In such cases, the value of variable $\theta_n^{M-1}(\alpha)$ is not constrained by equation (6-14) and is determined dynamically by the optimization procedure. By not imposing the restriction of equation (6-14) the solution space is increased by a factor of P which can in turn lead to an increased possibility of finding better scheduling solutions.

Scheduling computational paths for multilayer feed-forward neural network models, such as the multilayer perceptron network [66], can be performed in a number of different ways. We can attempt to exploit layer level parallelism by assigning an appropriately sized number of processors to each layer of the network and performing the optimization procedure for the assignment problem for each layer separately. Having a specific assignment of neurons to processor and having different processors associated with neurons of each layer, the creation of computational paths is simply performed by having path α terminate at a processor holding the output value of a neuron in the next higher layer of the neural network. Although simple in nature, this approach is not effective for implementations on 2-D connected parallel architectures, such as the DREAM Machine, since all paths must cross a 1-D boundary between processors assigned to neurons of layer ℓ and terminate in a region assigned to neurons of layer ℓ+1. On the other hand, the approach can be applied very effectively to architectures having higher dimensional interconnection structures, such as the 3-D computer of Hughes Research Labs [48].

The second option for implementing multilayer feed-forward networks, which was eluded to earlier, involves the use of relabeling processors as computation moves from one layer to the next. Computational paths can be constructed without any specific restriction on where each path is to begin or terminate for the first layer of the network. Having arrived at a scheduling solution, the processor where each path α terminates in is assigned the neuron of the next higher layer which is associated with path α. Unfortunately this technique might not be sufficiently efficient, in general, since only the assignment of neurons to processors of the first layer are done in an optimal fashion. In

order to remedy this problem, the assignment procedure of Section 6.3 can be generalized to cluster neurons in an optimum sense considering the multilayer structure of the neural network. Having a specific assignment of neurons to processors for each layer of the network, path α associated with a specific neuron of layer $\ell+1$ can be forced into terminating in the desired processor associated with that neuron. Thus, the scheduling problem can be done separately for each layer of interconnections in the neural network.

6.5 Constraint Nets Used for Optimization

By having an analytical formula for evaluating the *goodness* of a specific solution, either for the assignment problem given by equation (6-13) or the scheduling problem given by equation (6-29), many different techniques can be used to find such solutions [21, 33, 36, 37, 78]. The simplest method for finding optimal solutions for this minimization problem is to exhaustively search the solution space for the solution(s) with the lowest cost. Due to the extremely large solution space of our problem, such an approach is entirely impractical for all but the most trivial cases. Therefore, we need to consider other approaches, which do not guarantee the optimum solution but can generally find solutions close to optimum.

A direct method for performing the optimization task is to follow the gradient of the cost function in the direction of lowest cost value. This can be performed by evaluating the cost function of an arbitrary random starting state configuration and slightly modifying the configuration state such that the cost function is lowered. This process can be repeated until no slight modification of the system configuration can be found that would further minimize the cost function. This approach has a problem of getting stuck in local minimas in the cost function configuration space. We can expect a large number of such local minimas for complex problems such as the assignment and scheduling problems described above.

Probabilistic approaches to the optimization problem have been proposed to avoid the faith of being trapped in a local minima solution of high cost value. A well known method based on statistical mechanics for solving combinatorial optimization problem is the *Simulated Annealing* procedure [36]. With this technique, slight modifications to the configuration state are made which lower the cost function. These modifications are also

made even if it increases the cost function with a probability proportional to $e^{-(1/T)\Delta C}$, where T is analogous to the temperature of a physical system and ΔC is the change in the change in the cost function resulting from this new configuration. The optimization procedure begins at a high temperature where *uphill* jumps (system modifications causing increase in the cost function) are tolerated with a relatively high probability. The system temperature is then gradually lowered until a solution is found at $T \approx 0$. This process is slow and inherently serial in nature. Several extensions and variants of this approach, one of which is described later in this section, have been proposed to improve the convergence rate and allow for parallel implementations [77, 81].

Recently, several neural computation based techniques have been applied in solving combinatorial optimization problems [2, 13, 28, 33]. A major advantage of these methods over previous techniques is associated with their inherently parallel structure which can lead to high throughput efficient implementations. The use of neural network methods on solving complex optimization problems have been demonstrated by applying these methods to the classical Traveling Salesman Problem (TSP). Durbin and Willshaw [13] have proposed a geometrical based method for solving the TSP problem. In this approach an elastic path is placed on the 2-D surface containing the cities to be traversed. Adjustments are made to the various points on this elastic path in order to have the path pass through all the cites. A similar approach using the self-organizing feature maps neural network [37] has also been proposed in [2]. Hopfield and Tank [33] have also formulated a method for solving the TSP problem using neural computation techniques. Although each of these formulation seem to involve a different approach, they all share a common underlying principle.

It has been shown that statistical mechanics can be used as the basic underlying principle for these methods [77]. Viewing neural computation in this light, the difference between the Hopfield net approach [33] and the elastic net approach [13] can be seen to in the treatment of the constraint terms of the cost function. For example, the constraint of not allowing a salesman to be in two places at the same time was enforced *softly* by adding a penalty term to the cost function in the Hopfield and Tank formulation. This restriction was enforced *strongly* in the elastic net approach where physical analogy to the elastic band explicitly disallows such conditions to occur. A general approach for enforcing certain constraints strongly has been introduced via Constraint nets [78]. Constraint net optimization is based on *mean-field annealing* which is similar in concept

to the Simulated Annealing procedure described earlier in that it allows for limited movement in the direction of increasing the cost function in order to escape local minimas. On the other hand, mean field annealing manipulates average probability values and is formulated in a deterministic form, alleviating a need for random number generators.

In general, the configuration state of the system can be specified by a state vector $\boldsymbol{\eta}$ of n dimensions. Assuming a binary valued state vector, the number of possible configurations of the system can be thus calculated to be 2^n. In many applications, such as the assignment and scheduling problems, a significant portion of this space corresponds to illegal configurations. Given a specific cost function defined by $C[\boldsymbol{\eta}]$, we can define a certain probability distribution function over the possible configuration space. The Boltzmann distribution of the form

$$P_\beta[\boldsymbol{\eta}] = \frac{1}{Z_\beta} e^{-\beta C[\boldsymbol{\eta}]} \tag{6-30}$$

has been widely used by various neural network models including the Constraint net. In equation (6-30), the parameter β is the inverse of the thermodynamic temperature parameter defined as

$$\beta = \frac{1}{T}. \tag{6-31}$$

A primary goal of the optimization process is to evaluate the what is called the partition function Z_β defined as

$$Z_\beta = \sum_{\{\boldsymbol{\eta}^*\}} \exp\left(-\beta C[\boldsymbol{\eta}^*]\right), \tag{6-32}$$

where $\{\boldsymbol{\eta}^*\}$ refers to the set of all the 'legal' configuration of the state vector $\boldsymbol{\eta}$. An important property of this partition function is that at zero temperature the partition function is dominated by the configuration with the lowest cost, that is

$$Z_\beta \underset{\beta \to 0^+}{\approx} e^{-\beta C[\boldsymbol{\eta}^0]}, \tag{6-33}$$

where $\boldsymbol{\eta}^0$ refers to the configuration with the lowest cost.

As mentioned earlier, this formulation is termed mean-field annealing since the state vector variables used in the calculation refer to the average probability of events at a specific temperature given by

$$\langle \eta \rangle_\beta = \frac{1}{Z_\beta} \sum_{\{\eta^*\}} \eta e^{-\beta C[\eta^*]}. \tag{6-34}$$

By using equation (6-34) in conjunction with equation (6-32) we can calculate the configuration with the lowest cost ($\boldsymbol{\eta}^0$). The major difficulty in implementing this procedure involves performing the sum operation of equation (6-32) only over the legal configurations specified by $\{\boldsymbol{\eta}^*\}$. In the Constraint net formulation presented in [78], a method is demonstrated which uses an *effective cost function* to analytically approximate $\langle \eta \rangle_\beta$. In constructing this effective cost function, we can choose to eliminate none, some, or all of the illegal configuration in the summation operation of the partition function. The constraints which are explicitly removed from this sum are said to be *strongly* enforced; those which are not, are said to be *softly* enforced. The convergence properties of this method has been proven analytically in [78].

6.6 Optimization Procedure

In Section 6.4, we presented the cost function used for solving the assignment problem. We assumed at the time that all variables where binary valued. In Section 6.5, we saw how mean-field annealing can be used to solve optimization problems. Following the formalism of Section 6.5, we can treat each variable as being continuous valued representing an average probability of an event. In other words, the value of variable $\tilde{\omega}_n^A$ indicates the probability that neuron n is mapped to processor A. Similarly, the variable $\tilde{\eta}_A$ indicates the probability that processor A is assigned a neuron, $\tilde{\theta}_{n(\alpha)}^i$ represents the probability that path α is at a processor assigned neuron n at time i, and so on.

In this section, we present the method used for solving the cost optimization problem based on the Constraint net technique. This method is applied to both the assignment and scheduling cost functions. The optimization process involves an iterative procedure for updating the values of the state variables ω_n^A and η_A of the assignment problem, and $\theta_{n(\alpha)}^i$, $\eta_{n(\alpha)}^i$, and ε_n^i of the scheduling problem. The formulation of the update equations for each of these variables, along with the specific procedure for their implementation, is given below.

6.6.1 Update Equations for the Assignment Problem

Update equations for the assignment variable ω_n^A can be arrived at using the Constraint net approach described in Section 6.5. The effective cost function for the assignment problem is given by equation (6-13). In deriving the update equations, we can strongly enforce the constraint of equation (6-5) requiring that each process (neuron) be assigned to some processor of the architecture. The update equation for each of the ω_n^A is written as

$$\delta\tilde{\omega}_n^A = -\delta t\left(\tilde{\omega}_n^A - \frac{e^{-\beta\tilde{\phi}_n^A(\tilde{\omega})}}{\sum_{B=0}^{P-1} e^{-\beta\tilde{\phi}_n^B(\tilde{\omega})}}\right), \tag{6-35}$$

where $\phi_n^A(\tilde{\boldsymbol{\omega}})$ is the first derivative of the cost function of equation (6-19) taken with respect to the variable ω_n^A. The value of each $\phi_n^A(\tilde{\boldsymbol{\omega}})$ is given by

$$\begin{aligned}\phi_n^A = &\frac{1}{4}\sum_{A,B}\sum_m\left(-G^{AB}\mathbb{G}^{AB}\mathbb{D}^{nm}\omega_m^B + \left(1-G^{AB}\right)\mathbb{G}^{AB}\mathbb{D}^{nm}\omega_m^B\right)\\ &+\frac{1}{2}\sum_m d_{nm}\omega_m^A + \frac{1}{2}\sum_B \gamma_{AB}\omega_n^B + \gamma_\omega\eta_A\left(\sum_{m=0}^{N-1}\omega_m^B - \omega_n^B - \frac{1}{2}\right).\end{aligned} \tag{6-36}$$

Due to the fact that there are no penalty terms associated with the variable η_A that can be enforced strongly, the update equation associated with this variable follows the basic form of the neurons in a Hopfield net [33]. The update equation for this variable is defined as

$$\delta\eta_A = -\delta t\left(\eta_A - \frac{e^{-\beta\tilde{\phi}_A(\boldsymbol{\eta})}}{1+e^{-\beta\tilde{\phi}_A(\boldsymbol{\eta})}}\right) \tag{6-37}$$

which has the familiar form of the sigmoid activation function with its gain parameter being controlled by β. At high temperatures, β is small and the sigmoid has a smooth and rather flat shape, corresponding to similar η_A values for different processors. As the temperature is lowered, the gain of the sigmoid function for each variable η_A is increased further, making each variable move closer to one of the two binary points (0 or 1), depending on the value of $\tilde{\phi}_A(\boldsymbol{\eta})$ defined by

$$\phi_A = \gamma_\eta \left(\sum_{B=0}^{P-1} \eta_B - \eta_A - N + 1 \right)$$
$$+ \frac{1}{2} \gamma_\omega \left(\left(1 - \sum_{n=0}^{N-1} \omega_n^A \right)^2 + \omega_n^A \left(1 - \omega_n^A \right) \right). \qquad (6\text{-}38)$$

Equation (6-38) is obtained by differentiating the effective cost function of equation (6-13) with respect to η_A.

6.6.2 Update Equations for the Scheduling Problem

Update equations for modifying the variables associated with the scheduling problem can be similarly derived using the Constraint net technique. We presented a general description of the scheduling cost function and how different constraints on the start and end location of each path effect the mapping method. The update equations associated with path variable $\theta_n^i(\alpha)$ can be written to specifically enforce none, some, or all of these special cases. The general form of the update equation is

$$\delta \tilde{\theta}_n^i(\alpha) = -\delta t \left(\tilde{\theta}_n^i(\alpha) - \frac{e^{-\beta \tilde{\phi}_n^i(\alpha)}}{\sum_{m=0}^{N-1} e^{-\beta \tilde{\phi}_m^i(\alpha)}} \right), \qquad (6\text{-}39)$$

where

$$
\begin{aligned}
\tilde{\phi}_n^i(\alpha) = & \frac{1}{2}\sum_{m=0}^{P-1} T_{nm}(\boldsymbol{\omega})\left(1-D^{\alpha m}\right)\tilde{\theta}_m^{i+1}(\alpha) \\
& + \frac{1}{2}\sum_{m=0}^{P-1} T_{nm}(\boldsymbol{\omega})\left(1-D^{\alpha n}\right)\tilde{\theta}_m^{i-1}(\alpha) \\
& + \frac{1}{2}D^{\alpha n}\sum_{j=0}^{i-1}\tilde{\theta}_n^j(\alpha) + \frac{1}{2}\sum_{j=0}^{M-2} D^{\alpha n}\tilde{\theta}_n^{j+1}(\alpha)\sum_{\ell=0}^{j}\delta^{\ell i} \\
& + \frac{1}{2}\eta_n^{i-1}(\alpha)D^{\alpha n}\left(1-\sum_{j=0}^{i-1}\tilde{\theta}_n^j(\alpha)\right) - \frac{1}{2}\sum_{j=0}^{M-2}\eta_n^j(\alpha)D^{\alpha n}\tilde{\theta}_n^{j+1}(\alpha)\sum_{\ell=0}^{j}\delta^{\ell i} \\
& + \frac{1}{2}\lambda_\infty\left(\sum_{m=0}^{P-1}\left(1-T_{nm}(\boldsymbol{\omega})\right)\tilde{\theta}_m^{i+1}(\alpha) + \sum_{m=0}^{P-1}\left(1-T_{mn}(\boldsymbol{\omega})\right)\tilde{\theta}_m^{i-1}(\alpha)\right) \\
& + \sum_{m=0}^{P-1} d_{nm}^{\theta}\theta_m^i(\alpha) + \frac{1}{2}\gamma_p\left(1-2\theta_n^i(\alpha)\right) + \gamma_A\sum_{\beta=0}^{A-1}\left(1-\delta^{\alpha\beta}\right)\theta_m^i(\beta) \\
& + \frac{1}{2}\gamma_\theta\varepsilon_n^i\left(1-2\theta_n^i(\alpha)+2\left(\sum_{\beta=0}^{A-1}\theta_n^i(\beta)-1\right)\right)
\end{aligned}
\quad . \qquad (6\text{-}40)
$$

Equation (6-40) was obtained by taking the first derivative of the scheduling problem's effective cost function given by equation (6-29). Equation (6-40) can be simplified by collecting and reordering some terms to be of the form

$$
\begin{aligned}
\tilde{\phi}_n^i(\alpha) = & \frac{1}{2}\sum_{m=0}^{P-1} T_{nm}(\boldsymbol{\omega})\left(\left(1-D^{\alpha m}\right)\tilde{\theta}_m^{i+1}(\alpha) + \left(1-D^{\alpha n}\right)\tilde{\theta}_m^{i-1}(\alpha)\right) \\
& + \frac{1}{2}D^{\alpha n}\left(1-\eta_n^{i-1}(\alpha)\right)\sum_{j=0}^{i-1}\tilde{\theta}_n^j(\alpha) + \frac{1}{2}\sum_{j=0}^{M-2}\left(1-\eta_n^j(\alpha)\right)D^{\alpha n}\tilde{\theta}_n^{j+1}(\alpha)\delta_{(i\le j)} \\
& + \frac{1}{2}D^{\alpha n}\eta_n^{i-1}(\alpha) + \frac{1}{2}\lambda_\infty\sum_{m=0}^{P-1}\left(1-T_{nm}(\boldsymbol{\omega})\right)\left(\tilde{\theta}_m^{i+1}(\alpha) + \tilde{\theta}_m^{i-1}(\alpha)\right) \\
& + d^{\theta}\sum_{m=0}^{P-1}\left(1-\delta_{nm}\right)\theta_m^i(\alpha) + \frac{1}{2}\gamma_p\left(1-2\theta_n^i(\alpha)\right) + \gamma_A\sum_{\beta=0}^{A-1}\left(1-\delta^{\alpha\beta}\right)\theta_m^i(\beta) \\
& + \frac{1}{2}\gamma_\theta\varepsilon_n^i\left(1-2\theta_n^i(\alpha)+2\left(\sum_{\beta=0}^{A-1}\theta_n^i(\beta)-1\right)\right)
\end{aligned}
\quad , \qquad (6\text{-}41)
$$

where

$$
\delta_{(i\le j)} = \begin{cases} 1 & \text{if } i \le j \\ 0 & \text{Otherwise} \end{cases} . \qquad (6\text{-}42)
$$

The starting time (i=0) and the termination time (i=M-1) of the computational paths require slightly different equations. If we are to restrict the path α to return to the processor holding its neuron output value, such as in the cases of iterative neural networks, constraint of equation (6-14) can be satisfied by having ${\theta_n^{M-1}}_{(\alpha)} = \delta^{n\alpha}$ for all $\alpha = 0, 1, \cdots, A-1$. In this case, the value of ${\theta_n^{M-1}}_{(\alpha)}$ is not updated. Knowing that each path must end in the processor holding its neuron output value, we can further limit the partition function of the path variables associated with the next to the last time slot. We can strongly enforce that all paths must be in processors directly connected to the paths final destination at time M-2. This is accomplished by having the update equation for the M-2 time slot be equal to

$$\delta\tilde{\theta}_n^{M-2}{}_{(\alpha)} = -\delta t\left(\tilde{\theta}_n^{M-2}{}_{(\alpha)} - \frac{T^{\alpha n} e^{-\beta\tilde{\phi}_n^{M-2}{}_{(\alpha)}}}{\sum_{m=0}^{N-1} T^{m\alpha} e^{-\beta\tilde{\phi}_m^{M-2}{}_{(\alpha)}}}\right). \tag{6-43}$$

Equation (6-41) can still be used to evaluate $\tilde{\phi}_n^{M-2}{}_{(\alpha)}$ while considering ${\theta_n^{M-1}}_{(\alpha)} = \delta^{n\alpha}$.

Two options are available for handling the starting condition of computational paths. First, we can strongly enforce that all paths start at a processor holding a neuron output value required for that paths computation. This is a valid constraint since each path must cross these processors at some point along its trajectory. This restriction can be enforced similar to the case for the M-2 time slot by having the update equation of variable $\tilde{\theta}_n^0{}_{(\alpha)}$ be equal to

$$\delta\tilde{\theta}_n^0{}_{(\alpha)} = -\delta t\left(\tilde{\theta}_n^0{}_{(\alpha)} - \frac{D^{\alpha n} e^{-\beta\tilde{\phi}_n^0{}_{(\alpha)}}}{\sum_{m=0}^{N-1} D^{m\alpha} e^{-\beta\tilde{\phi}_m^0{}_{(\alpha)}}}\right). \tag{6-44}$$

By having such a formulation, we can speed up the convergence by leaving out all the unnecessary variables associated with the initial time slot. On the other hand, by restricting the possible solution space to only those solutions which start in a processor on the particular path, we might leave out certain globally minimum or close to globally minimum solutions. In case we would like to enforce the above condition softly, so that good solutions having some paths which start in processors outside their particular list of useful neurons can also be generated, the general form of the update equation (given by

equation (6-39)) can be used for $\tilde{\theta}^0_{n(\alpha)}$. The associated $\tilde{\phi}^0_{n(\alpha)}$ equation for this case can be written as

$$\begin{aligned}\tilde{\phi}^0_{n(\alpha)} &= \lambda_0\left(1-D^{\alpha n}\right)+\frac{1}{2}\sum_{m=0}^{P-1}T_{nm}(\boldsymbol{\omega})\left(1-D^{\alpha m}\right)\tilde{\theta}^1_{m(\alpha)}+\frac{1}{2}D^{\alpha n}\tilde{\theta}^1_{n(\alpha)}\\ &-\frac{1}{2}\sum_{j=1}^{M-2}\eta^j_{n(\alpha)}D^{\alpha n}\tilde{\theta}^{j+1}_{n(\alpha)}+\frac{1}{2}\lambda_\infty\sum_{m=0}^{P-1}\left(1-T_{nm}(\boldsymbol{\omega})\right)\tilde{\theta}^1_{m(\alpha)}\\ &+d^\theta\sum_{m=0}^{P-1}\left(1-\delta_{nm}\right)\theta^0_{m(\alpha)}+\frac{1}{2}\gamma_p\left(1-2\theta^0_{n(\alpha)}\right)+\gamma_A\sum_{\beta=0}^{A-1}\left(1-\delta^{\alpha\beta}\right)\theta^0_{m(\beta)}\\ &+\frac{1}{2}\gamma_\theta\varepsilon^0_n\left(2\sum_{\beta=0}^{A-1}\theta^0_{n(\beta)}-2\theta^0_{n(\alpha)}-1\right)\end{aligned} \quad , \qquad (6\text{-}45)$$

where λ_0 is the parameter associated with the soft constraint of having a path start in a non-useful processor.

In driving update equations for the $\eta^i_{n(\alpha)}$ variables, we can use some knowledge about the form of these variables to restrict the search space and increase the optimization performance. First we can notice that the variable $\eta^i_{n(\alpha)}$ participates only in those terms where $D^{\alpha n}=1$. Therefore, we can limit our computation of $\eta^i_{n(\alpha)}$ only to those α's and n's which have $D^{\alpha n}=1$. Next, we can observe from Figure 6-6 that the $\eta^i_{n(\alpha)}$ variables have a step function form where $\eta^0_{n(\alpha)}=0$ and $\eta^{M-1}_{n(\alpha)}=1$. The object of the $\eta^i_{n(\alpha)}$ update equations is thus finding the proper location where $\eta^i_{n(\alpha)}$ transitions from a zero to a one value. We can limit our search space here as well by choosing a properly formed partition function. Since there are only M-2 different valid configurations for $\eta^i_{n(\alpha)}$, the $\eta^0_{n(\alpha)}$ and $\eta^{M-1}_{n(\alpha)}$ are known and the rest of the configuration corresponding to different transition points along the time axis, the search space is reduced from 2^M to M-2.

We define a constant binary valued set of vectors $\boldsymbol{\eta}^{(d)}_{n(\alpha)}$ of size M-2 corresponding to valid $\eta^i_{n(\alpha)}$ configurations as

$$\boldsymbol{\eta}^{(d)}_{n(\alpha)} \equiv \left(\eta^{(d)1}_{n(\alpha)}, \eta^{(d)2}_{n(\alpha)}, \eta^{(d)3}_{n(\alpha)}, \cdots, \eta^{(d)M-2}_{n(\alpha)}\right). \qquad (6\text{-}46)$$

These vectors have the form shown below:

$$\begin{array}{lll} & 1\ \ 2\ \ 3\ \ 4 \qquad M\text{-}2 \\ \eta_n^{(1)}(\alpha) \equiv & (1,\ 1,\ 1,\ 1,\ \ldots,\ 1) \\ \eta_n^{(2)}(\alpha) \equiv & (0,\ 1,\ 1,\ 1,\ \ldots,\ 1) \\ \eta_n^{(3)}(\alpha) \equiv & (0,\ 0,\ 1,\ 1,\ \ldots,\ 1) \\ & \vdots \\ \eta_n^{(M-2)}(\alpha) \equiv & (0,\ 0,\ 0,\ 0,\ \ldots,\ 1) \end{array}$$

Given the set of vectors $\boldsymbol{\eta}_n^{(d)}(\alpha)$, we can define a partition function which is dominated by the $\eta_n^i(\alpha)$ vector producing the minimum cost at zero temperature. This partition function has the form

$$Z_n(\alpha) = \sum_{d=1}^{M-2} e^{-\beta \sum_{i=1}^{M-2} \tilde{\eta}_n^{(d)i}(\alpha)\phi_n^{(\eta)i}(\alpha)}, \tag{6-47}$$

where

$$\phi_n^{(\eta)i}(\alpha) = \frac{1}{2}\beta_\eta D^{\alpha n}\left(1 - 2\tilde{\eta}_n^i(\alpha)\right) + \frac{1}{2}D^{\alpha n}\tilde{\theta}_n^{i+1}(\alpha)\left(1 - \sum_{\ell=0}^{i}\tilde{\theta}_n^\ell(\alpha)\right). \tag{6-48}$$

The corresponding update equations for $\eta_n^i(\alpha)$ can now be derived to be

$$\delta\tilde{\eta}_n^i(\alpha) = -\delta t\left(\tilde{\eta}_n^i(\alpha) - \frac{1}{Z_n(\alpha)}\sum_{d=1}^{M-2}\eta_n^{(d)i}(\alpha)e^{-\beta\sum_{j=1}^{M-2}\tilde{\eta}_n^{(d)j}(\alpha)\phi_n^{(\eta)j}(\alpha)}\right). \tag{6-49}$$

Finally, the update equations for the ε_n^i variables can be derived. These variables, similar to η_A variables used in the assignment equations, do not have any strongly enforceable constraints. Therefore, the update equations associated with these variables are the simple Hopfield net equations (similar to equation (6-37)). The update equation is described by

$$\delta\tilde{\varepsilon}_n^i = -\delta t\left(\tilde{\varepsilon}_n^i - \frac{e^{-\beta\phi_n^{(\varepsilon)i}}}{1 + e^{-\beta\phi_n^{(\varepsilon)i}}}\right), \quad \text{where} \tag{6-50}$$

$$\begin{aligned}\phi_n^{(\varepsilon)i} &= \frac{1}{2}\gamma_\varepsilon\left(1 - 2\tilde{\varepsilon}_n^i + 2\left(\sum_{m=0}^{P-1}\tilde{\varepsilon}_m^i - A\right)\right) \\ &\quad + \frac{1}{2}\gamma_\theta\left(\left(1 - \sum_{\alpha=0}^{A-1}\tilde{\theta}_n^i(\alpha)\right)^2 + \sum_{\alpha=0}^{A-1}\tilde{\theta}_n^i(\alpha)\left(1 - \tilde{\theta}_n^i(\alpha)\right)\right).\end{aligned} \tag{6-51}$$

6.6.3 Implementing the Optimization Procedure

In the previous section, we presented the update equations associated with the various variables involved in the assignment and scheduling optimization procedures. The actual processing of the optimization procedure is described here. Since the optimization algorithm is based on mean-field annealing, where the values of each variable refer to the probability of an event, we start the system with all variables initialized to small random values. The range of these random numbers are calculated differently for different variables, depending on their specific constraints. For example, the assignment variables $\tilde{\omega}_n^A$ are assigned a random value in the range (0, 2/*P*) so that the sum over all processors (*A*'s) of these variables will approach 1, as *P* approaches infinity, therefore conforming to the constraint of equation (6-5).

Along with initializing the variables, the system temperature is set to an initial temperature value. This initial temperature must be large enough (higher than the system's *critical temperature*) so that the optimization procedure can move out of small local minima's of the cost function. After this initialization phase, the update equations are applied to all the variables *Nsteps* many times, after which the system temperature is lowered by one or two percent. This processing continues until the system converges to a stable state at low temperatures. At this point the solutions to the assignment and scheduling problems can be obtained from the close to binary valued matrices $\tilde{\omega}$ and $\tilde{\theta}$, respectively.

As part of the optimization procedure, the scaling parameters associated with the various penalty terms of the cost functions, such as λ_∞ and γ_{AB} are increased as a function of the temperature. Typically, the values of these penalty terms are increased such that they are infinitely large at T=0. In our implementation, we divide the initial parameter value by the square-root of the temperature to achieve this effect.

Chapter 7

Results of the Optimization Procedure

In this Chapter we present empirical results of the optimization procedure applied to several examples. These results where obtained through software simulation of the Constraint net (Cnet) algorithm described in the previous chapter. We first present a description of the implementation code used to perform the optimization task, followed by results of executing the assignment and scheduling optimization routines on some benchmark examples.

7.1 Implementation of the Optimization Algorithm

Two separate routines have been developed to solve the assignment and scheduling optimization problems. The basic organization and computational flow of both the assignment and scheduling routines are similar. The first step of each routine involves reading a control file, containing various file names and control parameter associated with the specific optimization routine, followed by the execution of the optimization algorithm, with appropriately set parameter values, for a specified number of sweeps. Each sweep consists of a number of steps the update equations are applied to each variable at a constant temperature value.

7.1.1 Control File Format

The control file associated with each routine contains the names of the various input and output files associated with the process graph file, interprocessor connection topology

graph, etc. Additionally, the various parameters associated with each routine, such as d_{nm}, γ_{AB}, and λ_{∞}, along with system temperature and cooling parameters are also given in the control file. The control files associated with the assignment and scheduling routines are shown in Tables 7-1 and 7-2. The control parameter *Nruns* denotes the number of times the optimization routine should be executed, each with a different initial random state.

Nruns	- Number of runs
N	- Number of neurons
A	- Number of paths
P	- Number of processors
$D^{\omega n}$	- Name of neural net structure file
G^{AB}	- Name of architecture topology file
ω_n^A	- Name of the assignment output file
δt	- Learning rate
T_0	- Initial Temperature
ΔT	- Temperature mod. rate
Nsweeps	- Number of sweeps
Nsteps	- Number of steps/sweep
d_{nm}	- Penalty parameter of equation (6-6).
γ_{AB}	- Penalty parameter of equation (6-7).
γ_{ω}	- Penalty parameter of equation (6-11).
γ_{η}	- Penalty parameter of equation (6-12).

Table 7-1 - Control file format used for the assignment routine

Nruns	- Number of runs
N	- Number of neurons
A	- Number of paths
P	- Number of processors
$D^{\alpha n}$	- Name of neural net structure file
G^{AB}	- Name of architecture topology file
ω_n^A	- Name of the assignment matrix file
M	- Maximum time value to complete all paths
δt	- Learning rate
T_0	- Initial Temperature
ΔT	- Temperature mod. rate
Nsweeps	- Number of sweeps
Nsteps	- Number of steps/sweep
λ_0	- Penalty parameter of equation (6-18).
$\gamma^{\alpha\beta}$	- Penalty parameter of equation (6-23).
γ_p	- Penalty parameter of equation (6-28).
γ_θ	- Penalty parameter of equation (6-26).
γ_ε	- Penalty parameter of equation (6-27).
λ_∞	- Penalty parameter of equation (6-22).
d_{nm}^θ	- Penalty parameter of equation (6-25).
β_η	- Penalty parameter of equation (6-21).

Table 7-2 - Control file format used for the scheduling routine

7.1.2 The Assignment and Scheduling Optimization Routines

The basic processing associated with the optimization procedure was outlined previously in Section 6.6.3. Both the assignment and scheduling routines follow similar control flows. They differ in the computation of the update equations and the actual variables being optimized. The pseudocode representation of the assignment problem optimization routine is given as:

```
1. Read control file information.
2. Read process graph file D^αn.
3. Read architecture topology graph G^AB.
4. Calculate the distance matrix G^AB based on G^AB.
5. Initialize ω and η_A matrices to small random values.
6. Initialize system temperature (T ← T_0)
7. Repeat the following for Nsweeps times.
     7.1 Repeat the following Nsteps times.
        7.1.1 Update ω and η_A matrices according to
        equations (6-35) and (6-37).
     7.2 Decrease the system temperature (T ← (ΔT)T).
     7.3 Adjust Penalty term parameters.
8. Write final ω and η_A matrices to the output file.
```

The pseudocode of the scheduling routine is represented as:

```
1. Read control file information.
2. Read process graph file D^αn.
3. Read architecture topology graph G^AB.
4. Calculate the neuron connection graph T_nm according to
   equation (6-16).
5. Initialize θ, η, and ε matrices to small random
   values.
6. Initialize system temperature (T ← T_0)
7. Repeat the following for Nsweeps times.
     7.1 Repeat the following Nsteps times.
        7.1.1 Update θ, η, and ε matrices according to
        equations (6-39), (6-46), and (6-50).
     7.2 Decrease the system temperature (T ← (ΔT)T).
     7.3 Adjust Penalty term parameters.
8. Write final θ, η, and ε matrices to the output file.
```

The software used for the assignment optimization routine has been developed on a Sun workstation using the C programming language. This software takes as input 3 ASCII files, namely the control file, the process graph file, and the network topology file. It generates the assignment matrix ω and stores it in an ASCII file. During each sweep of processing, the various portions of the assignment cost value is calculated and displayed. In order to collect statistically valid information on the convergence properties of the algorithm as a function of the parameter values, a data-parallel implementation of this code was executed on the 512 node Ncube parallel machine at Caltech. In this implementation, each node executed the same algorithm with different initial states on

different processors. This type of parallelism was defined earlier as network level parallelism for neural network implementations, see Chapter 2.

The scheduling routine has been implemented both on the Sun workstation using the C programming language, and also on the AMT DAP-610C parallel processor using the DAP FORTRAN-Plus parallel programming language. The DAP architecture consists of a 2-D mesh-connected array of bit-serial processors. The DAP-610C model contains 4096 PEs with each PE having an 8-bit arithmetic coprocessor. Due to the parallel nature of the matrix operation associated with the Cnet algorithm, close to linear speedup was achieved by implementing the code on the DAP.

7.2 Results of the Assignment Procedure

We now present some empirical results obtained from executing the assignment optimization routine on two examples. The first example problem has been proposed by Bokhari [7] as a benchmark for evaluating the performance of his heuristic algorithm for solving the assignment problem. The second problem is a larger benchmark proposed by Everstine [14] to be used for comparing the performance of various algorithms in solving sparse matrix bandwidth, profile, and wavefront reduction algorithms.

7.2.1 Results from Implementing the Bokhari Example

Two specific examples have been used by Bokhari to measure the performance of his algorithm for solving the assignment problem [7]. Both examples attempt to assign the nodes of a specific process graph, obtained from a finite element matrix, to the processors of a special purpose parallel architecture, called the Finite Element Machine (FEM). The first example involves mapping a 33 node process graph onto a 36 processor FEM. The objective of Bokhari's algorithm is to perform the assignment operation such that the number of edges in the process graph which are assigned to the edges of the processor graph is maximized. Bokhari refers to this measure as the *cardinality* of the assignment. This objective function can be viewed as a subtask of our assignment objective function, since in addition to maximizing the amount of overlap between the two graphs (assignment cardinality), we require specific treatment of nodes

which are not assigned to their optimum location. Below, we present a detailed description of the 33 node example along with the performance results of our algorithm in solving it. The results associated with the 25 node example are briefly described at the end of this section.

7.2.1.1 Problem Description

The target machine architecture used for the Bokhari example is the FEM. The FEM architecture consists of a 2-D array of processors connected in eight-nearest-neighbor configuration with wrap-around connections on the two boundaries (top-to-bottom and left-to-right). A 6x6 FEM is used as the target architecture for this example problem, see Figure 7-1. The interconnection topology of this architecture can be represented in the form of an interconnection matrix G^{AB} required by our optimization algorithm, see Figure 7-2.

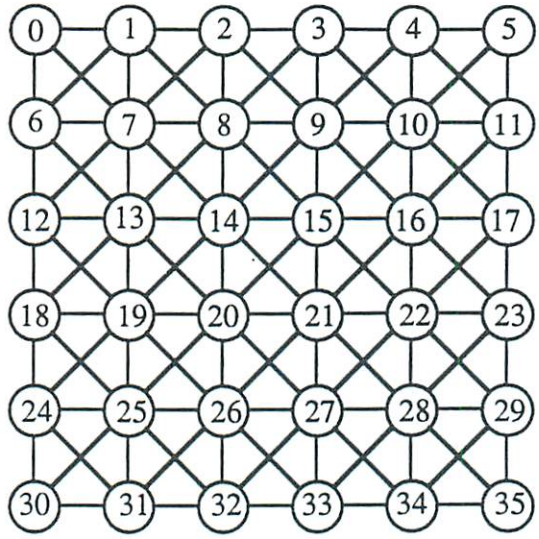


Figure 7-1 - Interconnection topology of a 6x6 Finite Element Machine (FEM).

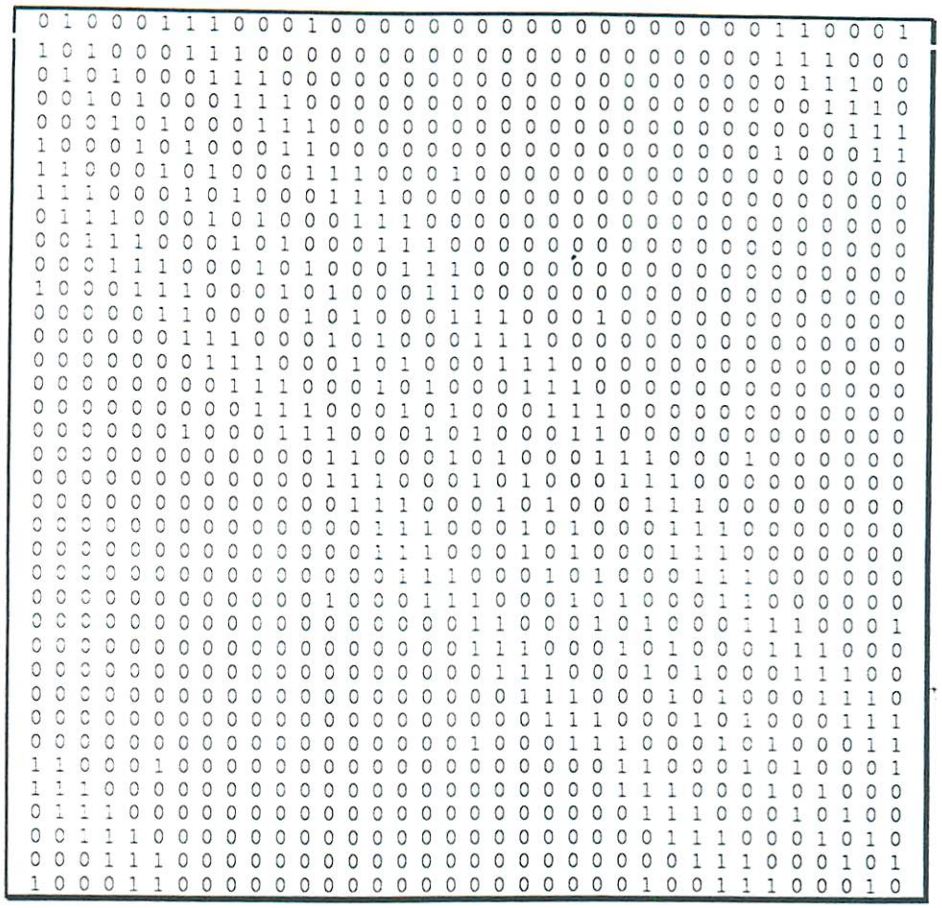


Figure 7-2 - The interconnection matrix G^{AB} describing the 6x6 FEM topology.

The process graph used for this example is a 33 node finite element structure shown in Figure 7-3, with the corresponding dependence graph representation D^{α_n}, shown in Figure 7-4. The G^{AB} and D^{α_n} files are given as input to our optimization procedure. The assignment procedure generates an appropriate distance matrix $\mathcal{G}^{AB}$ based on the specified G^{AB} file, see Figure 7-5. Since the number of nodes in the process graph is smaller than the number of nodes in the processor graph, some processors are not assigned any process.

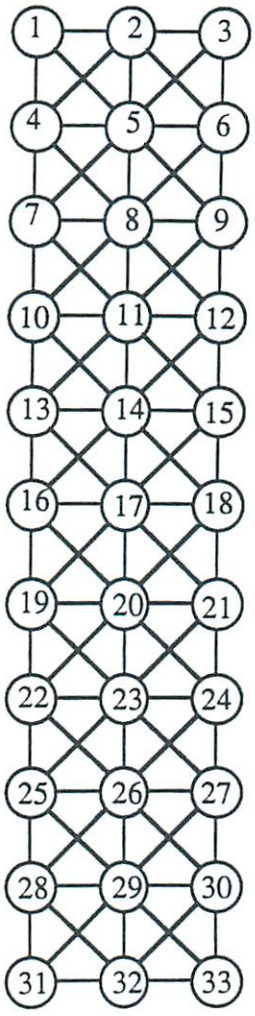


Figure 7-3 - A 33 node finite element structure used as the process graph.

The process graph of Figure 7-3 contains a total of 80 edges. This means that the best possible assignment would have a cardinality of 80. Due to the inherent differences between the two graphs, such an assignment does not exist [7]. The best achievable cardinality for this problem is reported to be 78 [7]. A cardinality value of 32 is achieved with a simple linear assignment of process i to processor i.

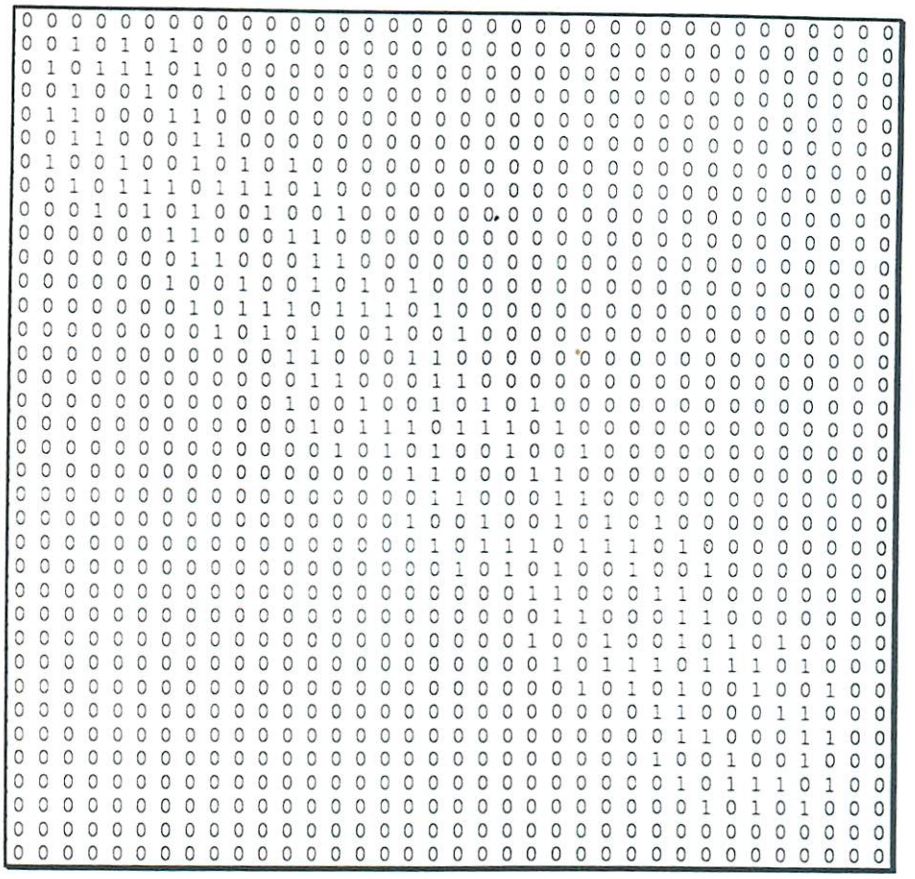


Figure 7-4 - The process graph interconnection matrix D^{α_n}.

The optimization procedure was executed many times to collect statistical information on the performance of our algorithm for different parameter values. Based on these experiments, we have determined a generally good range of parameters for the assignment optimization routine. Table 7-3 gives some statistical results of our simulations. The columns titled "Max_c" and "Min_c" indicate the maximum and minimum cardinality values found in all runs, respectively. The column titled "% Opt" indicates the percentage of runs where the globally optimum solution was found. The columns titled "Mean" and "SD" indicate the mean and standard deviation of the cardinality values, respectively. The last row of this table indicates the results of a Monte-Carlo search performed for this assignment problem. This implementation involved generating a random assignment matrix ω and calculating the cardinality of this solution. Figure 7-6 shows a histogram of the Monte-Carlo experiment with 1000 points.

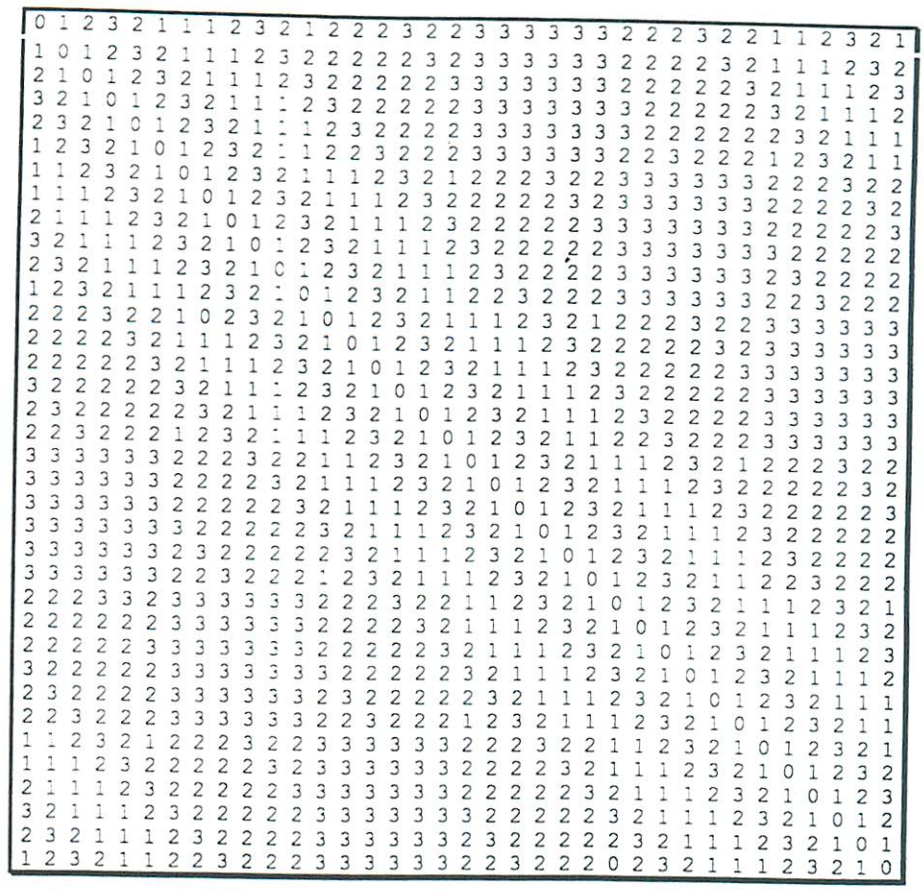


Figure 7-5 - Interprocessor communication distance matrix $\mathcal{G}^{AB}$ obtained from the architecture topology graph G^{AB}.

Nruns	γ_{AB}	γ_{ω}	γ_{η}	Max_C	Min_C	% Opt	Mean	SD
1231	0.0	0.5	0.0	78	48	30.5	70.92	8.29
60	0.5	0.4	0.15	78	64	33.3	75.6	3.15
44	0.5	0.4	0.1	78	65	25.0	75.9	2.61
1000	0.0	0.0	0.0	30	8	0.0	18.1	3.65

Table 7-3 - Statistic collected from executing the assignment optimization routine on the 33 node Bokhari example. The shaded row is the results of a Monte-Carlo search algorithm used for comparison.

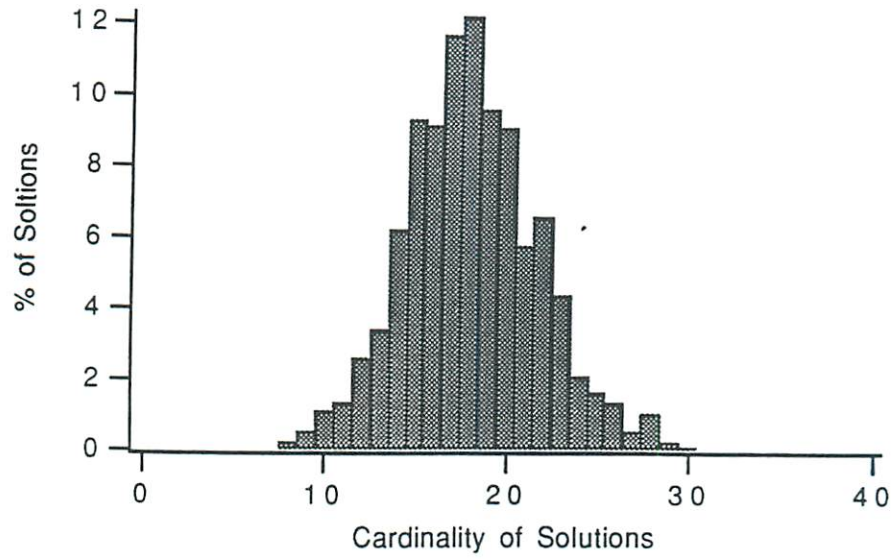


Figure 7-6 - The Histogram associated with 1000 randomly generated assignments for the 33 node Bokhari example.

In all the runs included in Table 7-3, the other parameter values, not shown in that table, had the following values:

$$\begin{aligned} T_0 &= 2.0 \\ \Delta T &= 0.99 \\ \delta t &= 0.025 \\ d_{nm} &= 2 - \mathbb{D}_{nm} \end{aligned}$$

In the first batch of runs (with 1231 runs), equation (6-7) used to discourage assigning the same process to two different processors and equation (6-12) used to find processors participating in the mapping, where not implemented. Also, in these runs, the constraint for assigning a single process to each processor was enforced according to equation (6-9) rather than the more selective equation (6-11). It can be noticed that the other runs shown in Table 7-3, demonstrate solutions with much improved mean cardinality values and smaller variances in their cardinalities.

From Table 7-3, we can notice that in general, about 1/3 of the runs end in finding the globally optimum solution. Figure 7-7 depicts a histogram associated with the experiment of size 1231 runs. It is interesting to notice the presence of several local minimas distributed across the cost function landscape. The convergence characteristics of the assignment optimization routine is presented graphically in Figures 7-8 through

7-10. From these figures, we can see that the network has converged after approximately 200 sweeps at a temperature around T=0.4. It is interesting to note, from Figures 7-9 and 7-10, that all of the cost values begin a sharp decline at a critical temperature around 0.5. At this temperature, structure is being formed from the seemingly random distribution of probabilities in the assignment matrix ω.

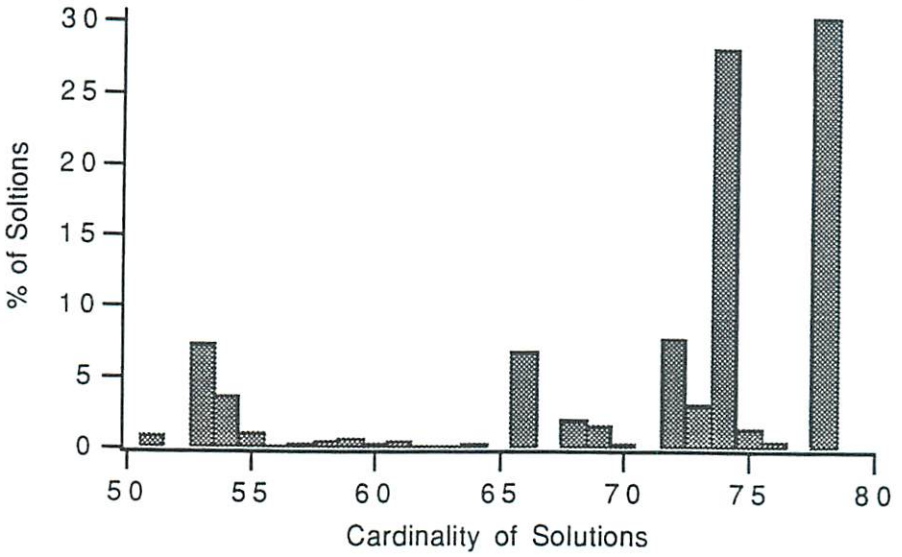


Figure 7-7 - Histogram of solution cardinalities for the 1231 runs described in Table 7-3. The global optimum solution is at cardinality 78 which is found the largest percentage of times.

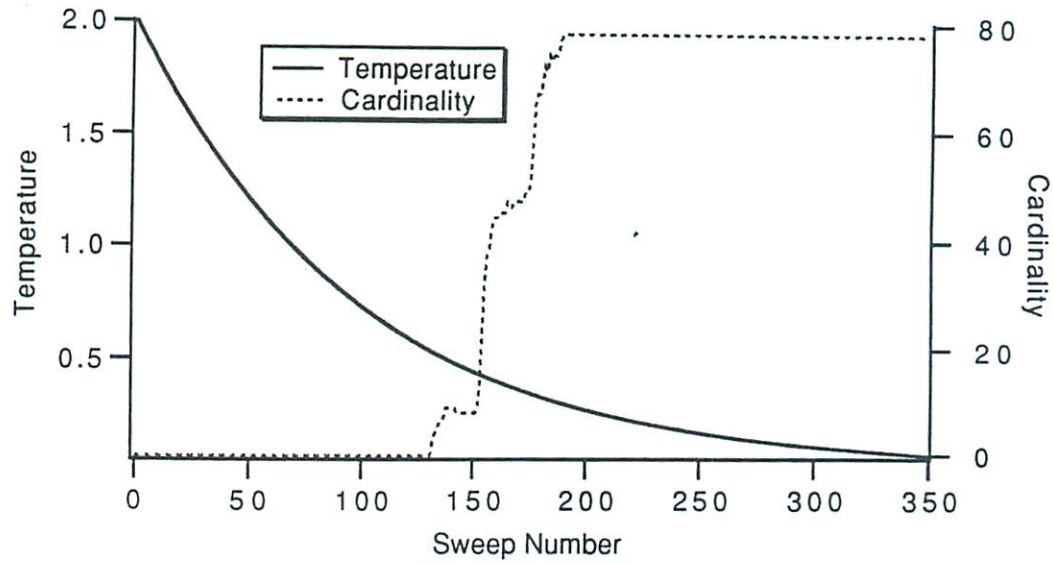


Figure 7-8 - Graph of system temperature and cardinality value vs. sweep number.

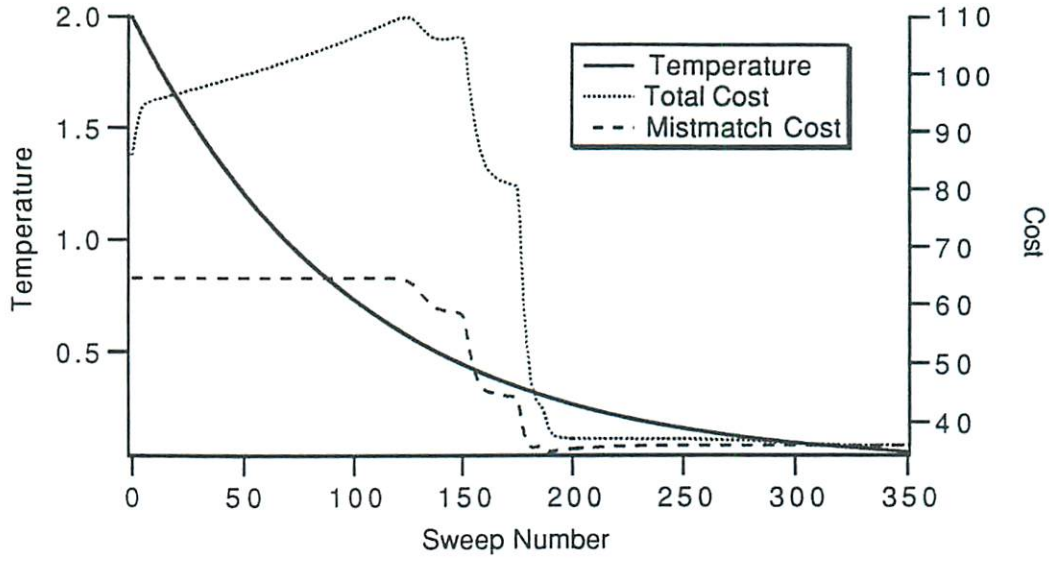


Figure 7-9 - Graph of system temperature, total cost (given by equation (6-13)), and mismatch cost (given by equation (6-1)) vs. sweep number.

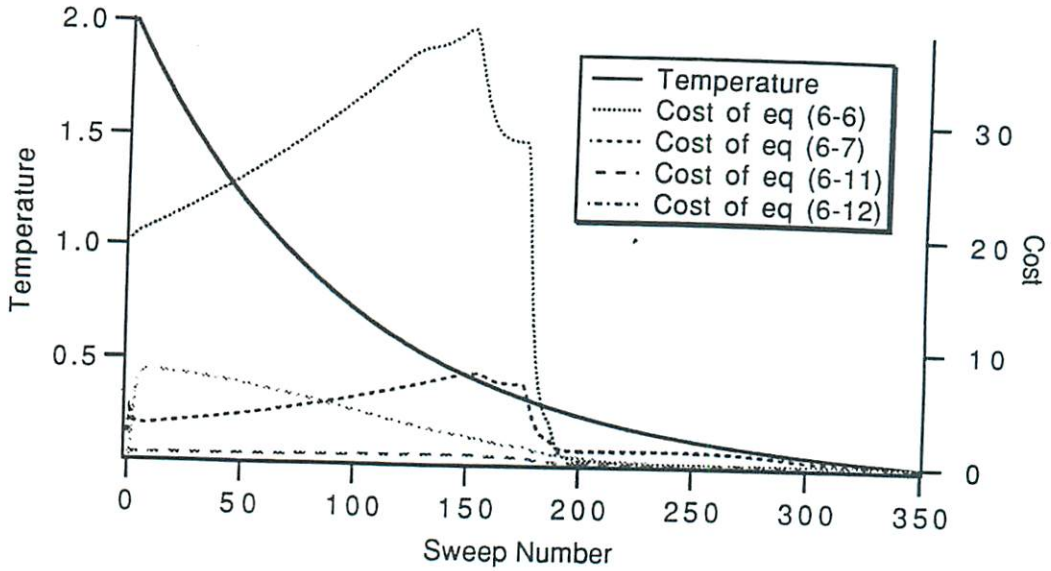


Figure 7-10 - Graph of system temperature and the various penalty terms (given by equations (6-6), (6-7), (6-11), and (6-12)) vs. sweep number.

A limited number of experiments have also been performed on another example proposed in [7] involving the mapping of a 25 node process graph to a 5x5 FEM processing array. A histogram of our results with non-optimal parameter settings is given in Figure 7-11.

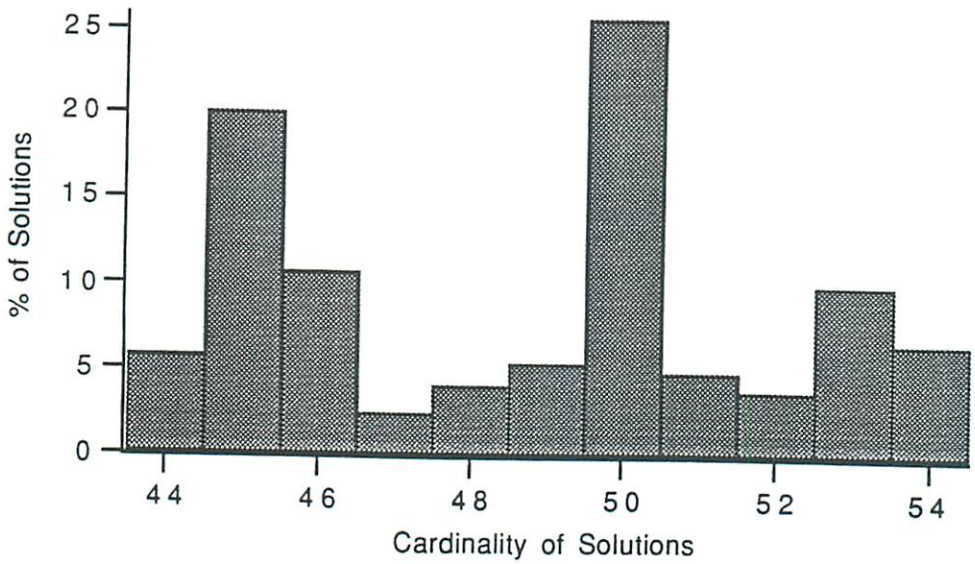


Figure 7-11 - Histogram of the solution cardinalities when mapping the 25 node process graph to FEM of size 5x5 taken from a collection of 736 runs.

In general, our results compare very well to that obtained by Bokhari. Using the algorithm described in [7], the best assignment found by Bokhari had a cardinality of 74. Comparing the computational complexity of both algorithms, the Bokhari algorithm requires $O(N^3)$ operations. Our algorithm, on the other hand, requires $Nsweeps*O(P^2N)$ operations. Asymptotically the computational complexity of both algorithms are similar. Our Cnet approach seem to be superior to the Bokhari technique in both finding better solutions and having a deterministic parallel structure.

7.2.2 Results from Implementing the Everstine Example

The assignment optimization routine has also been applied to a larger size problem, proposed by Everstine [14]. The problem involves a sparsely connected finite element graph with 59 nodes and 104 edges having an interconnection density of only 7.67%. This problem has originally been proposed as a benchmark for evaluating the performance of various algorithms on reducing the matrix bandwidth and profile attributes. The matrix bandwidth problem is closely related to the assignment problem since both attempt to find an optimum relabeling [7]. For the assignment problem, this relabeling is performed on the nodes of the process graph, and for the matrix bandwidth reduction problem it is performed on the row and column indices of the matrix.

The target architecture used for this assignment is a 2-D array with 64 processors organized in an 8x8, 4-nearest neighbor interconnected topology. Similar to the Bokhari example, we have performed a 1000 Monte-Carlo experiment to evaluate the difficulty of this particular assignment problem. Due to the low density of the processes graph (7.67%), random assignments give very poor cardinalities having an average of 6.5 out of the theoretical maximum of 104. Figure 7-12 shows the distribution of the cardinality values obtained from our Monte-Carlo experiment as a histogram.

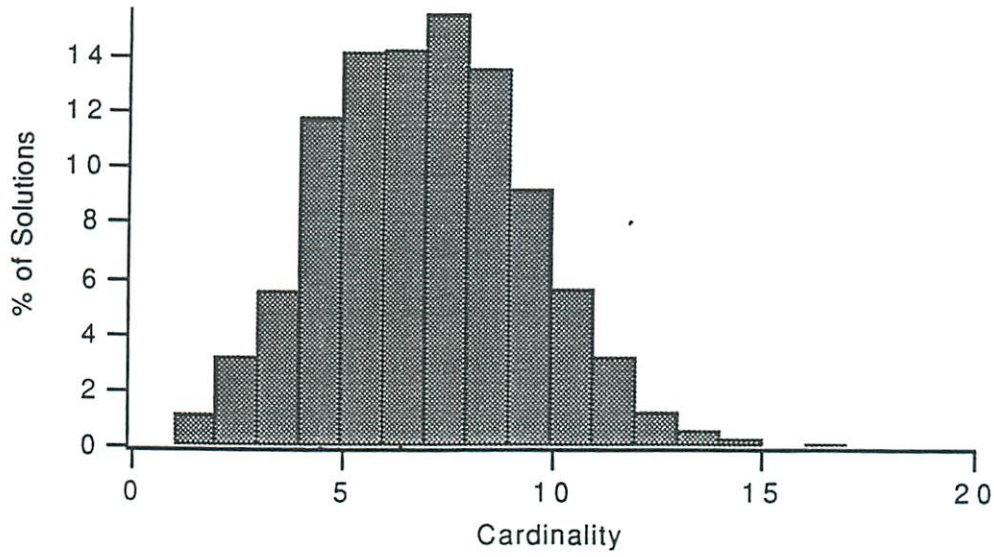


Figure 7-12 - The histogram associated with solution cardinalities of 1000 random assignments of the Everstine 59 node example mapped to a 4-nearest neighbor connected 8x8 parallel processing architecture.

Due to the excessively large search space ($64!/5! \approx 10^{87}$), it is not possible to exhaustively search the solution space for the optimum assignment giving the largest possible cardinality. However, we can see from Figure 7-12 that finding good solutions is difficult. The original process graph D^{α_n} used in this problem is fairly banded with a bandwidth of 26 and a profile of 464. Note: the bandwidth of a matrix is defined to be the maximum of all the individual row bandwidths of a matrix, where a row bandwidth is defined as the number of columns between the first non-zero element of the row up to and including the diagonal. The profile of a matrix refers to the sum of all the individual row bandwidths of the matrix.

Due to the existence of this initial structure in the D^{α_n} matrix, a linear assignment of nodes from the process graph to nodes of the processor graph yield a fairly high cardinality value of 42. Statistical data on several runs performed using our assignment optimization algorithm and the Monte-Carlo algorithm are presented in Table 7-4. Since no optimum solution is known for this problem, it is difficult to know how close to the optimum are these solutions. Of course, we can compare the results with randomly generated solutions and also to the fairly well structured initial solution.

Nruns	γ_{AB}	γ_{ω}	γ_{η}	Max_C	Min_C	Mean	SD
25	0.1	0.5	0.15	72	63	68.56	2.24
13	0.05	0.4	0.15	73	65	68.69	2.10
1000	0.0	0.0	0.0	16 .	0	6.50	2.45

Table 7-4 - Statistics collected from executing the assignment optimization routine on the 59 node Everstine example. The shaded row is the results of a Monte-Carlo search algorithm used for comparison.

Note that in all the runs included in Table 7-4, the parameter values not shown in that table, had the following values:

$$\begin{aligned} T_0 &= 1.5 \\ \Delta T &= 0.98 \\ \delta t &= 0.025 \\ d_{nm} &= 2 - \mathbb{D}_{nm} \end{aligned}$$

7.3 Results of the Scheduling Procedure

In order to evaluate and demonstrate the use of our scheduling optimization routine, we have selected to apply the algorithm to two different problems. Similar to the problems selected for the assignment optimization task, one of the problem contains a known globally optimum solution and our task is to measure the difference between the solutions generated by our procedure against this optimum. The second problem involves a slightly more complicated scheduling with no known optimum solution. However, as mentioned in Section 6.4.3, there is a theoretical upper bound on the optimum solution, namely the number of neurons in the neural network N.

7.3.1 Receptive-Field Example

The first example solved by our scheduling optimization problem involves finding non-conflicting paths associated with the computation necessary to implement the neural network structure shown in Figure 7-13, on a 16 processor parallel architecture with a

4x4 4-nearest neighbor interconnection topology. This particular neural network is structured in a commonly used receptive field, or convolution type, form. The number of neurons used for the mapping is 16, referring to the number of neuron in the input layer. There are four separate computational paths, one associated with each path α, which pass through the various PEs concurrently. Each path must traverse 9 neurons assigned to 9 processors arranged in a 3x3 local neighborhood.

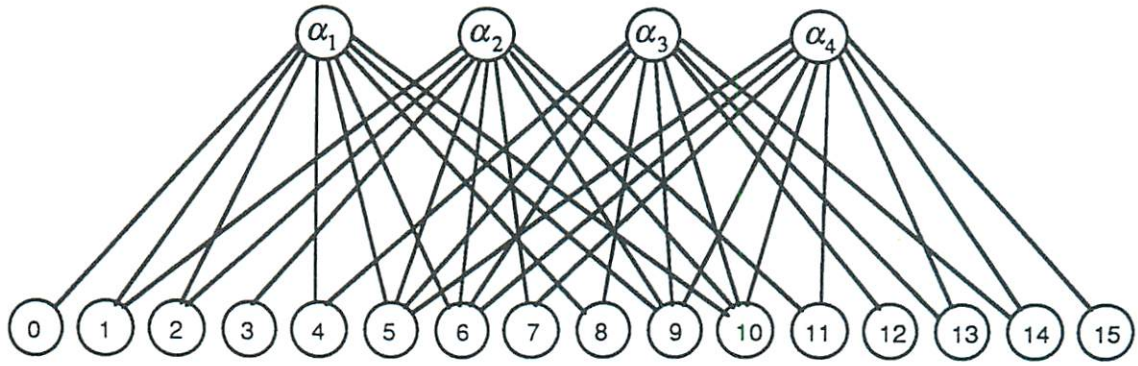


Figure 7-13 - Neural network with a receptive field type interconnection structure.

It is a well known technique to generate computational paths associated with this type of a structure by having each path traverse in a *whirlpool* shape. Each path is offset in space by one or more processors. With this type of a scheduling, the maximum required time to complete all paths is equivalent to the number of neurons associated with each receptive field (in this case 9). Therefore the global optimum scheduling will have all paths complete their traversal of the required neurons in 9 cycles.

The upper bound on the number of cycles need to complete all traversals is equal to the number of neurons in the first layer of the network, namely 16. Therefore, we began our optimization procedure with the time limit parameter M set to 11. This means that we require all paths to be of length 11, and all the neurons required for the computation of a particular path must be traversed within 11 cycles. Figure 7-14 illustrates the flow pattern of the four paths over the 4x4 processor array. In all of our runs we manually selected the center processor in each receptive field to be the terminating location of its associated computational path. For example, processor number 5 is selected as the terminating point of path α_1.

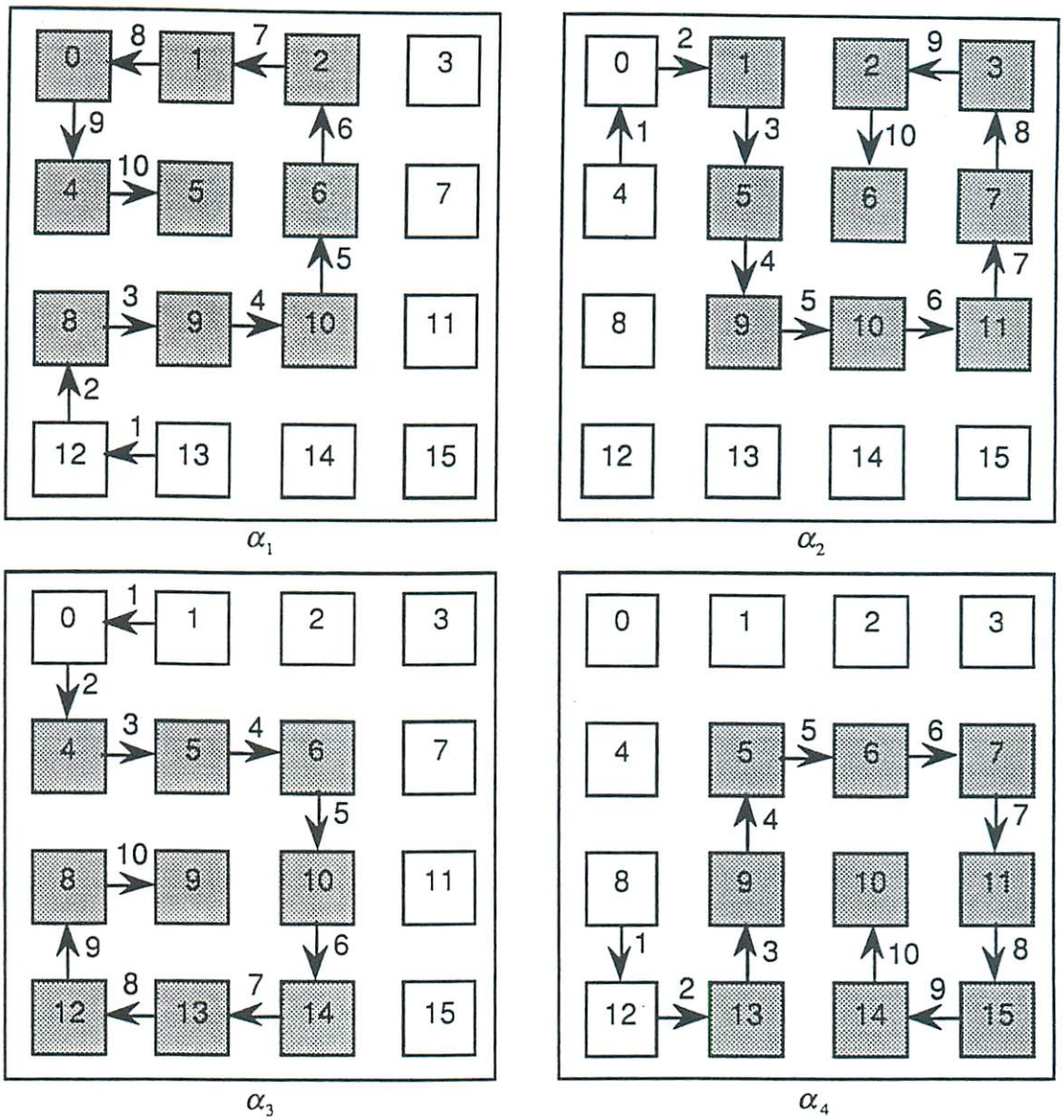


Figure 7-14 - Scheduling flow pattern for the 4 paths associated with Figure 7-13. The shaded boxes indicate the required neurons to be traversed by each path.

It is interesting to note that all the paths shown in Figure 7-14 start in processors not required for the computation of the associated path. Nevertheless, since we have required that each path be of length M=11, the paths cover unnecessary neurons. It is easy to see from Figure 7-14, that path traversals are limited to communications between neighboring processors. It is assumed that interprocessor communication can be performed in full duplex mode, that is two neighboring processors can exchange data

concurrently. The DREAM Machine architecture allows for such communication through the use of two X-net switches. With a close examination of these paths, we can show that no two paths ever cross the same processor at the same time, therefore satisfying our scheduling constraints. The parameter setting associated with the run shown in Figure 7-14 are as follows:

$$
\begin{aligned}
T_0 &= 0.25 \\
\Delta T &= 0.992 \\
\delta t &= 0.03 \\
\lambda_0 &= 1.0 \\
\lambda_\infty &= 1.0 \\
\gamma_A &= 0.2
\end{aligned}
$$

The parameter values associated with the remaining penalty terms of the scheduling cost function where set to zero and where eliminated from the optimization computation for all the examples in this section.

Figure 7-15 shows a different flow pattern generated with the same value of M=11. In this figure, we can notice that the paths choose to stay in a processor during one communication cycle (cycle 8). The apparent symmetry in the flow patterns generated in both Figures 7-14 and 7-15 is due to the symmetrical nature of the problem and not necessarily a consequence of our optimization algorithm. This symmetry disappears when scheduling an asymmetrical problem, such as the one presented in Section 7.3.2. The parameter values used for this run where:

$$
\begin{aligned}
T_0 &= 0.25 \\
\Delta T &= 0.99 \\
\delta t &= 0.03 \\
\lambda_0 &= 1.0 \\
\lambda_\infty &= 0.5 \\
\gamma_A &= 0.2
\end{aligned}
$$

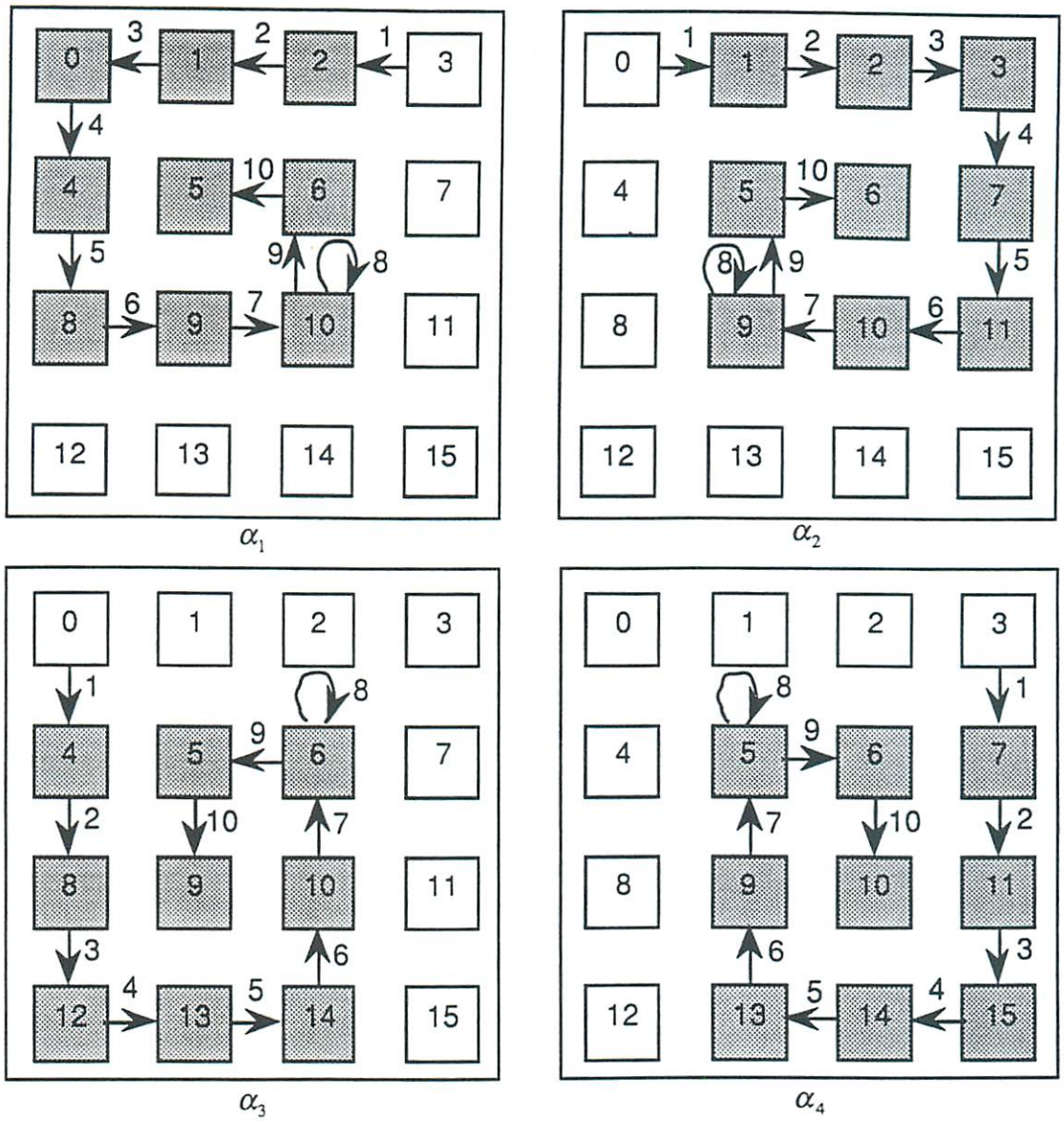


Figure 7-15 - A second scheduling flow pattern for the 4 paths of Figure 7-13, with path length M=11. The shaded boxes indicate the required neurons to be traversed by each path. A loop in the path indicates that the path stayed in the processor for the corresponding cycle.

With good results obtained with path length M=11, we further reduced M to the global optimum point of 9 cycles. We where able to consistently find one of the many whirlpool shaped flow patterns that can be constructed to cover all the required 9 neurons for each path. An example of one such solution is shown in Figure 7-16. Again, we can

notice an interesting symmetry in these flow pattern resulting from the restrictions imposed on the path traversals.

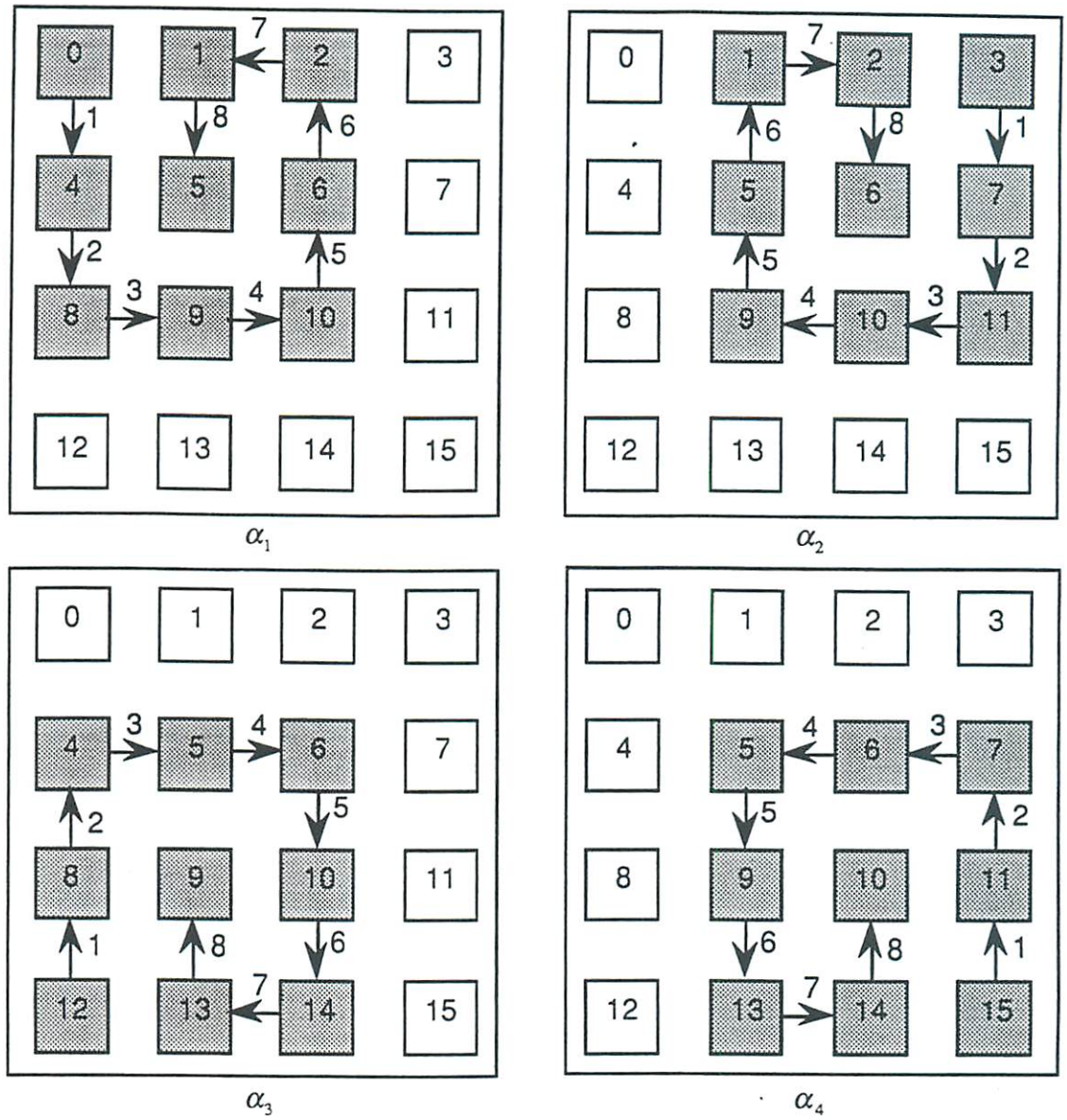


Figure 7-15 - Scheduling flow pattern for the 4 paths of Figure 7-13, with optimum path length (M=9).

From these examples we can see that although the task of finding solutions that satisfy the complex constraints of our scheduling problem is difficult, the enormous size of the solution space offers many solutions which might not be optimal but be close enough to achieve considerable speedup over the simple regular flow patterns.

7.3.2 Randomly Sparse Matrix Example

In addition to solving the scheduling problem for regularly structured interconnection networks, such as the receptive field structure used in the previous section, we have analyzed the performance of our scheduling optimization algorithm on randomly interconnected structures. In this section we describe an example implementation involving a sparse iterative system with 16 neurons. The interconnection network structure is shown in Figure 7-16, where a filled circle at a specific row and column indicates a connection between the corresponding neurons. The target machine architecture used for this mapping is similar to the one used for the receptive field example. Namely, a 2-D 4-nearest neighbor connected 4x4 processing array, except the target machine here is assumed to have wrap-around connections between the top and bottom and between the left and right side processors.

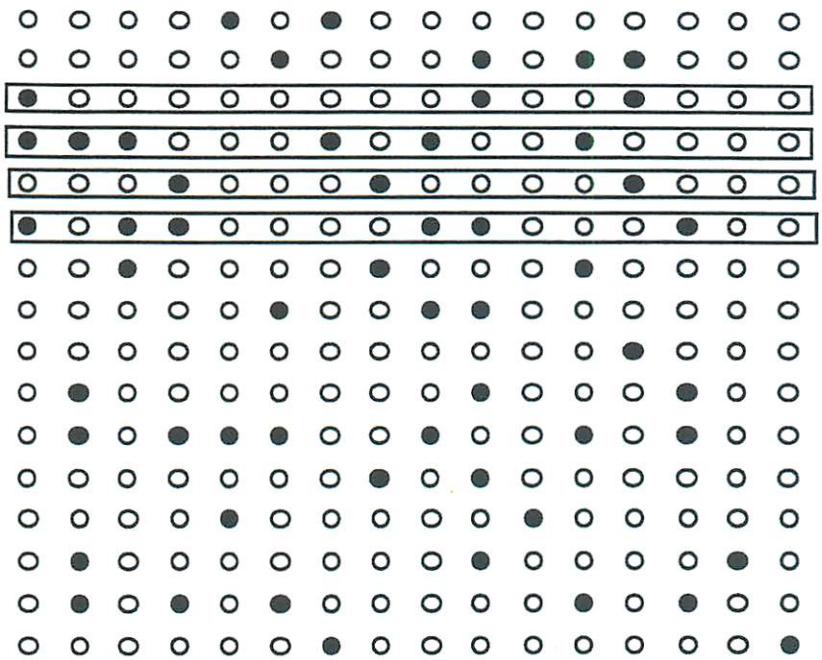


Figure 7-16 - Interconnection matrix associated with a sparse iterative system.

In order to simplify the scheduling problem we must first assign neurons participating in the same computational path to processors close to one another using the assignment optimization procedure. We have chosen not to perform this assignment operation and perform the simple linear assignment of neurons to processors. In this way, we have

increased the difficulty in finding good solutions and can better predict the performance of our algorithm on larger and more complicated interconnection structures.

Since the neural network being implemented is iterative, such as the Hopfield model [33], we strongly enforce constraint (6-14) for having each path α terminate in the processor holding its associated neuron output value. For example, the path associated with the computation of neuron 1 must terminate at the processor having been assigned neuron 1.

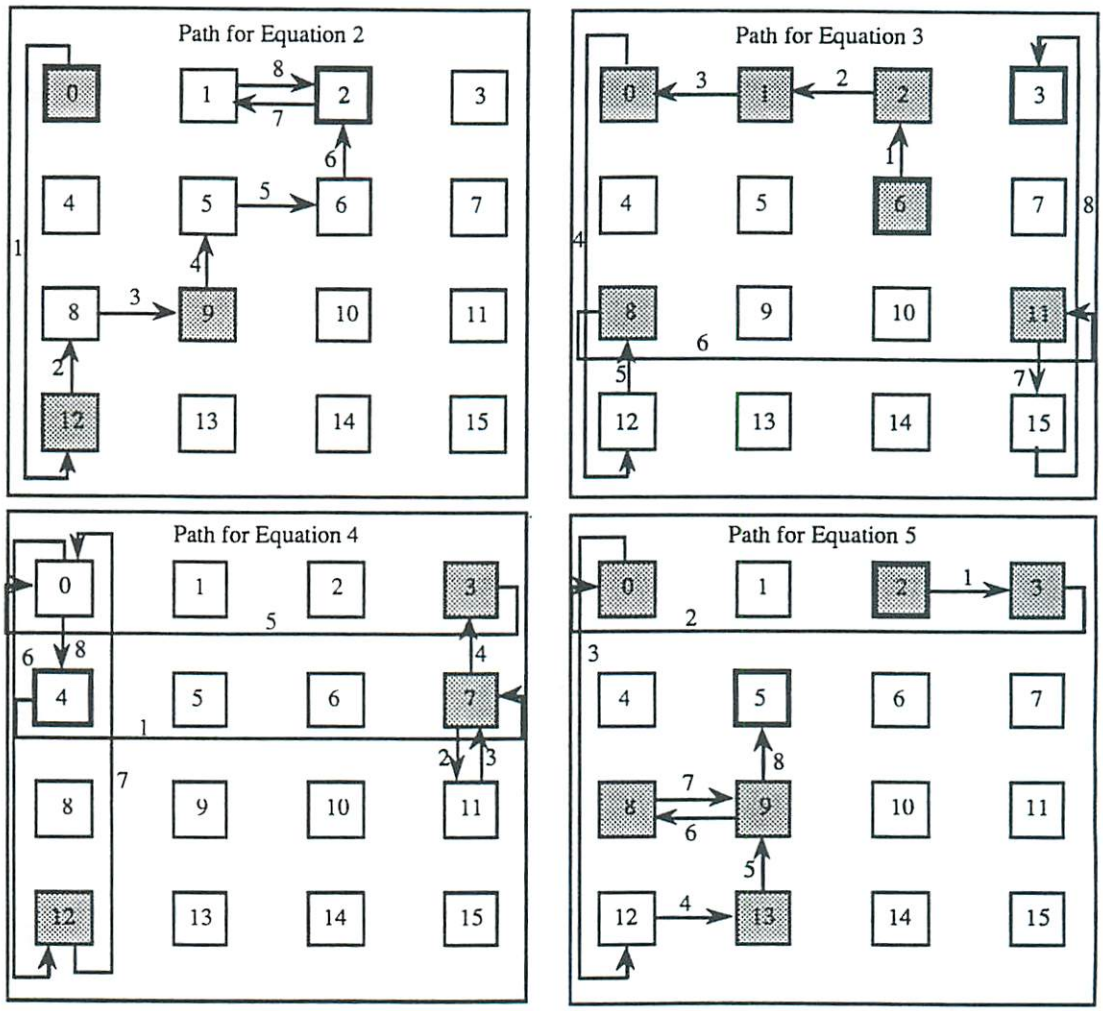


Figure 7-17 - Four of the 16 paths associated with the sparse iterative system shown in Figure 7-16. The shaded boxes indicate neurons which must be traversed by the associated paths. The boxes with thick outlines indicate start and end processors of each path.

The optimal path length M^* for implementing the network structure shown in Figure 7-16, is bounded below by K^*=8 being the maximum number of neurons to be traversed by a single path, including itself, and bounded above by the total number of neurons in the network N=16. In Figure 7-17 we show the flow patterns for four paths, one associated with each of the boxed rows of Figure 7-16. In this run we have set the path to be of length M=9. We can now see that due to the irregular interconnection pattern of the $D^{\alpha n}$ matrix (Figure 7-16), there is no regularity in the flow patterns of each path.

We can see that all paths have traversed their required processors and there seem to be room for further reduction in the path length parameter M. This process can be repeated several time to find a satisfactory solution to the scheduling problem.

Chapter 8

Asymptotic Performance Analysis

In this chapter, we present the asymptotic performance of our mapping method and the DREAM Machine architecture as applied to neural network processing. This analysis is beneficial from two aspects. First, asymptotic performance analysis can help to determine the performance of an implementation as the problem size and machine size increase. This is specially useful for neural network processing systems which are likely to increase in size by several orders of magnitude in the near future. The second benefit of this analysis is in enabling us to effectively compare various implementation methods. Although this analysis does not generate exact execution rates, it does offer a means to compare different implementations based on their scaling and relative cost factors.

The analysis is divided into three sections. First we describe the implementation complexity of neural network processing on the DREAM Machine. This is followed by a description of the computational complexity of our optimization based mapping method. Finally, we compare the asymptotic performance of our method with other previously proposed methods.

8.1 Implementation Complexity of Neural Network Processing

In Chapter 2 of this dissertation, we described the different parallel structures available in neural networks, that can be utilized to increase the implementation throughput. Moreover, we described the applicability of each type of parallelism for different neural network models. From these descriptions, we concluded that neuron level parallelism,

assigning single neurons to individual processors, is a good compromise solution offering both ample flexibility in type of models which can be implemented, and also by offering a large degree of parallelism for achieving high speed implementations.

8.1.1 Mapping Method Comparisons

We can simplify our analysis by considering neural computation as vector-matrix operations. In Chapter 2, we separated neural computation into local and distributed operations. An implementation based on neuron level parallelism is assumed to be able to perform local operations in constant time, $O(1)$. The distributed operations, involving commutative operations distributed across the processors of the machine, can be performed efficiently in one of two ways. Lets consider the distributed computation of the weighted input sum associated with a single neuron a in a neural network receiving inputs from N_a different neurons. This computation involves performing N_a addition operations, assuming an adder with two inputs and one output. It is known that this operation can be performed optimally in logarithmic time $O(\log N_a)$ using a binary tree topology.

In this implementation, parallelism is attained by performing multiple portions of the summation operation, associated with a single neuron, in parallel. Considering the simple case of a fully connected neural network the addition operation must be performed for all the N neurons in the network. Sequential processing of these operations, using binary tree type addition, will lead to an effective execution rate of $O(N \log N)$, with processors being idle $\log N$ percent of the total time. We may use pipelining techniques to increase the system efficiency by executing $\log N$ (the depth of the binary tree) neurons concurrently. With this method, the execution rate of the complete network will be $O(N)$, not considering the pipeline filling and flushing time.

The second method that can be used to efficiently implement the distributed computation of neural networks is to perform the weighted input sum sequentially. In this method, each processor adds its contribution to a partial sum value and passes the updated partial sum value to the next processor. This implementation method was described in detail in Chapter 4 and was shown to require $O(N)$ time to complete the calculation associated with a single neuron. In addition, we have demonstrated how the computation of N such neurons, in a fully connected neural network structure, can be

performed in parallel using a ring connected processing array. Hence, this method, similar to the binary tree approach, has a time complexity of $O(N)$.

A similar method to that of the ring-connected implementation involves broadcasting the neuron output values sequentially over a global communication bus [25]. This method has been employed in a commercially available neural network processing architecture. The performance characteristics of this method is inherently similar to the ring connected approach, except the time required for the complete computation is $O(N)$ regardless of the number of processors P. Therefore, in analyzing the performance of various implementations we only consider the ring connected approach with an understanding that the results can be directly applicable to implementations based on the broadcasting approach, since we assume $P=N$.

There are several reasons why the ring connected approach is more appropriate than the binary tree approach for implementing the distributed operations of a neural network. First, the number of interconnections required by the tree topology is 3 connections per processor, whereas the number of connections per processor of a ring topology is 2. Second and more importantly, iterative neural computation, such as those used in a Hopfield net [33], require that the result of the computation associated with a specific neuron be stored in the local memory of the processor assigned to the neuron. This is simply accomplished in the mapping method of Chapter 4 using the ring structure. However, this operation involves more data transfer cycles to implement on the tree architecture.

It should be noted here, however, that more complex interconnection topologies, such as the hypercube, can be utilized to implement the binary tree style addition operation when implementing sparse neural networks and also used as a ring when implementing dense neural network models. Due to the poor scaling characteristics of such interconnection topologies, requiring excessive amount of area and associated hardware for large systems, we will not include such approaches in our comparison.

Furthermore, in light of the above mentioned discussion, we can regard the mapping method of Chapter 4, to be optimum for implementing fully connected neural network structures. Therefore we will now consider the time and area complexity of neural computation for this mapping method as applied to neural network structures with arbitrary interconnection structures. In particular, we consider here the specific mapping

methods of variable-length ring and optimization based mappings, described in Chapters 5 and 6, respectively.

8.1.2 Area-Time Complexity of Neural Network Implementation Methods

The time complexity associated with implementing neural network computation on ring systolic processing arrays was presented in Chapter 4. In Chapter 5, we showed how, by varying the length of the processing ring, high throughput rates can be achieved for implementing sparse but regularly connected neural networks. In Chapter 6, we removed the regularity condition imposed on the neural network interconnection structure for efficient implementations. In this section we analyze the time and area complexity of these methods and compare them with other methods proposed in the literature.

The time complexity of mappings onto fixed-size ring architectures where shown to be $O(P)$, where P is the number of processors in the ring, in Chapter 4. This time complexity is arrived at by considering that each computational path associated with each neuron traverses the entire ring in order to compute the weighted input sum of equation (2-2). On the other hand, we have shown that with block-connected neural network structures, computation associated with multiple blocks in the same layer can be performed in parallel on the DREAM Machine. Furthermore, the computation associated with each block has a time complexity of $O(R_\ell)$, where R_ℓ is the number of processors in the ring associated with layer ℓ. Therefore, the overall time complexity of the implementation, utilizing variable-length rings, is $O(\tilde{R})$, where $\tilde{R}$ is the size of the largest block in the neural network structure and defined as $\tilde{R} = \max_{\ell \in L, b \in B_\ell} \{R_\ell^b\}$.

8.1.2.1 Complexity of the Fixed-Size Ring Implementation

We can use the well known AT^2 efficiency metric [85] to compare the performance of variable-size ring and fixed-size ring architectures for neural network processing. The AT^2 measure is commonly used to compare various VLSI implementation. It is measured by multiplying the area complexity of a design by the square of the time complexity associated with a particular problem implemented on this system. The Area complexity of the fixed-size ring architecture is $O(P)$, assuming that a constant area is

required to implement each processor. The linearity of the area complexity of this architecture is due to its nearest neighbor interconnection topology which can be implemented with constant length wires. If we consider the size of the memory area used for holding the synaptic weights values at each processor, the total area complexity of this implementation will be $O(P+N^2)$. Assuming the processing array size to be of the same order as the neural network size, the area complexity can be written as $A=O(N^2)$. Earlier in Chapter 4, the time complexity of the implementation on fixed-size ring architecture was shown to be $O(N)$. Therefore, the AT^2 measure for this implementation is $O(N^4)$.

8.1.2.2 Complexity of the Variable-Length Ring Implementation

A similar analysis can be performed on the variable-size ring architecture implemented on the DREAM Machine. Each of the N neurons must allow for a maximum of $\tilde{R}$ synaptic weights to be stored in its local memory area. As such, we arrive at an area complexity $A=O(N\tilde{R})$, with a constant multiplicative factor several times larger than the fixed-size ring implementation. This extra area is required for the additional wires in the interconnection network and for the increased size of memory holding the data routing information. Nevertheless, both of these factors increase the area complexity by a constant factor and can be dropped in an asymptotic performance evaluation.

The time complexity associated with implementing neural networks on the DREAM Machine, using the variable-length ring method, is more difficult to evaluate. We have shown the time complexity of this method to be $O(\tilde{R})$ in Chapter 5. The value of $\tilde{R}$ is inherently related to the interconnection structure of the particular neural network being implemented. For fully connected neural networks $\tilde{R}=N$, thus the time complexity of this approach is less than or equivalent to that of the fixed-size ring approach, since $N \leq P$. As mentioned earlier, a basic premise of this dissertation is that large scale neural network models are sparsely interconnected having a relatively constant number of synaptic connections per neuron. With this assumption we can treat $\tilde{R}$ as a constant. The AT^2 metric associated with this implementation, assuming a constant size block structure is found to be $AT^2=O(N\tilde{R}^3)=O(N)$.

8.1.2.3 Complexity of the Optimization Based Implementation

Implementing neural network models on the DREAM Machine, following the optimization based mapping method described in Chapter 6, has a similar performance to that of the variable-length ring approach. However, the optimization based mapping method does not rely on any specific regularities in the interconnection structure, such as block structures, to attain efficient mappings. As mentioned earlier, the optimization based mapping method attempts to find mappings that can complete all the necessary computation associated with a particular neural network structure in M cycles. In Chapter 6, we mentioned that the value of M is bounded above by the number of neurons in the network N, and bounded below by the maximum number of connections to a neuron.

The value of M can be represented as $M = k\tilde{S}$, where $\tilde{S}$ is the maximum number of synaptic connections per neuron (a constant value) and k is a constant greater than one. It should be noted here that there is no guarantee that a constant valued k can be found in the general case. The value of k depends directly on the degree of the interprocessor communication topology and the neural network structure. Even with a moderately low dimensionallity in the interconnection topology of the architecture (e.g., eight nearest-neighbor connected), it is expected that a constant k value can be found for most neural network structures. Therefore, by following the argument of a constant number of synaptic inputs per neuron presented earlier, the time complexity of this implementation can be written as $O(M) = O(1)$, assuming M remains constant with respect to N and P.

The area complexity of this approach is similar to the variable-length ring method, since both methods use the DREAM Machine as the target architecture and both method follow the same basic execution principles. In the case of the optimization based mapping, each neuron must locally store M synaptic weight values, some of which might be zero. Therefore, the area complexity of this mapping is $O(NM)$. This leads to an AT^2 measure of $O(NM^2)$.

8.2 Complexity Analysis of the Optimization Based Mapping Method

The complexity analysis of implementing neural networks on the DREAM Machine presented in the previous section, was associated strictly with the implementation of the neural computation. In general, the mapping algorithm associated with the fixed-size ring architecture requires minimal amount of computation. The assignment of neurons to processors and scheduling the flow of data between processors is straight forward. Neurons are assigned linearly to processors and the data flows in a circle around the processing ring. The mapping for variable-length ring is a bit more complicated. For complex block structured networks, it requires a rather heuristic approach in order to align the multiple processing rings such that the computation can flow efficiently between the various layers of the neural network. On the other hand, the complexity of computation associated with the optimization based mapping method is rather significant and can be evaluated analytically. In this section we evaluate the computational complexity associated with the optimization procedures used for solving the assignment and scheduling problems.

8.2.1 Time Complexity of the Assignment Problem

The time required to implement the assignment optimization procedure can be evaluated by considering the pseudocode describing this algorithm given in Section 7.1.2. From this pseudocode, it can be seen that the compute intensive portion of the algorithm is associated with executing the update equations in step 7. It can be noticed that this operation involves a doubly nested loop of size *Nsweeps* and *Nsteps*. Within these loops is the specific update equation used to update the assignment matrix $\boldsymbol{\omega}$.

The time complexity associated with updating the entire $\boldsymbol{\omega}$ matrix can be derived by analyzing the update equations (6-35) through (6-38). Performing the updating of the neuron to processor selection matrix $\boldsymbol{\eta}$, according to equations (6-37) and (6-38), requires only $O(P)$ time. Implementing the update equations of the assignment matrix $\boldsymbol{\omega}$ according to equations (6-35) and (6-36) requires a number of matrix-matrix-product operations. This leads to a time complexity of $O(P^2N)$ assuming $P \geq N$. Therefore, the total time complexity of updating the assignment matrix is $O(P^2N) + O(P) = O(P^2N)$.

The complete algorithm performs this updating function *Nsweeps*Nsteps* number of times. In general, both *Nsweeps* and *Nsteps* values can be considered to be constant. Therefore, the time complexity of the complete assignment procedure can be given to be $O(P^2N)$. However, the constant multiplicative factor here can be quite large. A similar time complexity value has been reported by Bokhari [7] for his heuristic approach for solving the assignment problem.

8.2.2 Time Complexity of the Scheduling Problem

The time complexity of the scheduling optimization algorithm can also be evaluated based on the computational demands of the specific update equations associated with this problem. The major portion of the computation associated with this algorithm is attributed to the computation of the update values for the scheduling matrix $\boldsymbol{\theta}$ and its related variables $\eta_n^i(\alpha)$ and ε_n^i. The computation associated with updating the ε_n^i values requires $O(MN)$ operations, where M is the maximum path length value and N is the number of neurons in the network. The computation associated with updating the $\eta_n^i(\alpha)$ values can be implemented efficiently by considering only those variables which participate in the computation. Specifically, we only need to determine $\eta_n^i(\alpha)$ values which have a corresponding $D^{\alpha n} = 1$. This computation is further simplified by limiting the search space through the use of the $\boldsymbol{\eta}_n^{(d)}(\alpha)$ matrix, as described in Chapter 6. The overall computational complexity associated with updating the $\eta_n^i(\alpha)$ variables is $O(\tilde{S}M^2A)$, where $\tilde{S}$ is the maximum number of synaptic connections per neuron which has been assumed to be a constant. Therefore, we can write the time complexity of updating the $\eta_n^i(\alpha)$ values as $O(M^2A)$.

The computational complexity of implementing the update equations for the $\boldsymbol{\theta}$ matrix can be shown to be $O(P^2AM)$. Thus the total computational complexity associated with the scheduling problem is determined to be $O(P^2AM)+O(M^2A)+O(MN)$. We have already presented an argument as why the maximum path length parameter M can be considered to be a constant with respect to processing array size and the neural network size. With this assumption, the asymptotic computational complexity of the scheduling procedure is $O(P^2A)+O(A)+O(N)=O(P^2A)$, assuming N and P are of the same order.

8.2.3 Tradeoffs on the Use of the Optimization Procedure

The computational complexity of the optimization procedure can be rather significant, as described in the previous sections. The lower-bound of the computational complexity of the combined assignment and scheduling procedures is $O(P^2A)+O(P^2N)=O(P^2N)$, assuming that the number of paths is of the same order of magnitude as the number of neurons in the network. However, the inherently parallel structure of the optimization procedures can effectively be used to considerably reduce the execution time of these algorithms. If one uses the target machine of the mapping, with P processors, to implement the optimization procedure, the required implementation time will become $O(PN)$. Even with this parallel implementation, the computational demands of the algorithm are considerable and must be compared to the expected gain to be attained from their use.

In general, the optimization procedure should be used to generate efficient mappings for neural network structures which have no apparent regularity and can be considered to be static for a large number of iterations. Otherwise the mapping method based on variable-length processing rings might offer a less computationally demanding solution. We can analytically compare the use of these two approaches.

Considering a sparse and irregularly connected neural network structure, with N neurons and a maximum of $\tilde{S}$ synaptic connections per neuron, the time required to implement this neural network on a P processor DREAM Machine, using the variable-length ring approach, is $O(N)$. On the other hand, using the optimization based mapping method, we can expect an execution rate of $O(M)$, where $M=k\tilde{S}$ and k is a constant greater than one. The time attributed to the variable-length ring mapping procedure can be considered to be $O(1)$ since it only requires the construction of a processing ring of length N. On the other hand, the time requirement of the optimization procedures, not considering the parallel implementation, is $O(P^2N)$ with a rather large constant multiplicative factor. The complete time to execute r iterations of the neural network, including the mapping, is $O(rN)$ using the variable-length ring mapping and is $O(r)+O(P^2N)$ using the optimization based mapping.

Using the above values, we can approximate the size of r for which the optimization procedure might be beneficial to use. This is accomplished by having

$$rN = r + P^2N, \qquad \text{and} \tag{8-1}$$

$$r = \frac{P^2N}{N-1} \approx P^2. \tag{8-2}$$

Therefore, a good heuristic to use for determining when to use the optimization procedure is by using $r > P^2$ as a criteria. If the number of iterations the neural network is used is greater than the square of the number of processors used for its implementation, it would be beneficial to use the optimization procedure. Of course, this is based on the asymptotic assumptions of having very large N and P values.

8.3 Performance Comparison

In this section, we compare the implementation characteristics of our mapping method on the DREAM Machine architecture with those of others. This comparison is based on the performance of each implementation in processing neural networks with arbitrarily complex interconnection structures. The basis for this comparison is the AT^2 metric described in Section 8.1. In order to have a fair comparison, we examine only SIMD nearest-neighbor connected architectures used for neural network processing. These include the fixed-size systolic ring method proposed in [42], the linear array architecture with broadcast buses proposed in [25], and the algorithmic mappings onto mesh connected SIMD arrays described in [47, 61].

Table 8-1, shows the collected area, time, and AT^2 measures associated with each mapping method. In this table, N is the number of neurons in the network, E is to the number of non-zero synaptic connections in the network, $\tilde{R}$ is the length of the largest block of neurons in the network, and M is the maximum length of time required to complete all computational paths in the neural network found by the optimization procedure. As mentioned earlier, the performance characteristics of the fixed-size ring architecture and its associated mapping method [42] and that of the linear array with broadcasting of the neurons output values [25] are equivalent. However, the architecture proposed in [25] employs a mechanism which alleviates the need for storing zero valued weights. Thus, the area requirement associated with this architecture is $O(N + E)$.

	A	T	AT^2
Fixed-Size Ring [42]	$O(N^2)$	$O(N)$	$O(N^4)$
Broadcasting Method of [25]	$O(N+E)$	$O(N)$	$O(N^2+NE)$
Algorithmic Method of [47]	$O(N+E)$	$O(\sqrt{N+E})$	$O(N^3+N^2E)$
Variable-Length Ring	$O(N\tilde{R})$	$O(\tilde{R})$	$O(N\tilde{R}^3)$
Optimization Based Method	$O(NM)$	$O(M)$	$O(NM^3)$

Table 8-1 - Comparison of several implementation methods used for neural network processing.

With the assumption of having a constant maximum size limit on the number of interconnections to a neuron, we can modify the performance table and simplify some of the terms. The first simplification can be made by combining the variable-length ring method with the optimization based mapping method. In essence, the variable-length ring method is identical to generating optimum assignment and optimum scheduling of the information flow by using the regularities in the interconnection structure of the neural network. From this perspective, the maximum block size $\tilde{R}$ is equivalent to the maximum path length M. Thus, we can use either $\tilde{R}$ or M in our analysis.

Furthermore, we can calculate the total number of interconnections E in the neural network as

$$E = \sum_{n=1}^{N} S_n, \qquad \text{where} \qquad (8\text{-}3)$$

S_n is the number of connections associated with neuron n. Equation (8-3) can be rewritten as

$$E = N\overline{S}, \qquad \text{where} \qquad (8\text{-}4)$$

$\overline{S}$ is the average number of connections per neuron. By assuming a constant number of connections per neuron, we can treat $\overline{S}$ as a constant with respect to the neural network size. Therefore, the time complexity of the algorithmic mapping of [47] can be written as $O(\sqrt{N})$ and a corresponding area complexity of $O(N)$ is attained.

Similarly, we can assume again that $M = k\tilde{S}$, with k being a small constant. Incorporating this condition in our analysis, the time complexity of our method reduces

to $O(1)$ and the area complexity reduces to $O(N)$. Since the fixed-ring approach does not have any provisions to efficiently handle sparsely interconnected neural networks, its time and area complexity are not effected by our assumptions here. Table 8-2 comparatively shows the performance characteristics of each implementation incorporating our assumption of constant connections per neuron.

	A	T	AT^2
Fixed-Size Ring [42]	$O(N^2)$	$O(N)$	$O(N^4)$
Broadcasting Method of [25]	$O(N)$	$O(N)$	$O(N^3)$
Algorithmic Method of [47]	$O(N)$	$O(\sqrt{N})$	$O(N^2)$
Variable-Length Ring / Optimization Based Method	$O(N)$	$O(k)$	$O(N)$

Table 8-2 - Comparison of several implementation methods used for neural network processing, assuming a constant number of connections per neuron.

Chapter 9

Conclusion

9.1 Summary

One of the primary factors in the resurgence of neural networks has been attributed to the availability of cost effective parallel processing technology. This technology can enable neural networks to be applied to large and complex "real-world" applications. In Chapter 2, we described several different levels of parallelism, inherently available in neural network models, which can be used as basis for mapping neural computation onto parallel processing architectures. Although a great deal of parallelism is available in neural networks, efficiently harnessing this parallelism has been difficult, primarily due to the vast variety of neural network models and the complexity of their interconnection structures. We have presented several previously proposed methods for implementing neural network models on parallel architectures in Chapter 2. We have shown that most implementation methods are applicable to neural network models with a specific type of interconnection structure, such as being densely or sparsely interconnected.

In Chapter 3, we outlined the architectural design of the Dynamically Reconfigurable Extended Array Multiprocessor (DREAM) Machine. A general description of the architecture was presented along with a detailed description of various architectural features specifically designed to address neural network computational demands. The major aspect of the DREAM Machine architecture, which allows it to be used efficiently for implementing a wide range of neural network algorithms with arbitrary complex interconnection structures, is its ability to have each processor communicate

autonomously with one of its nearest neighbors. This autonomy has been achieved through the use of dynamically reconfigurable switches in the interprocessor communication network. Each switch has a local data routing memory area which is accessed during every communication cycle. The contents of this routing table determine which of the communication channels are to be used by the particular processor for send and receive operations. By allowing each processor to uniquely communicate with any of its neighbors, complex flow patterns can be constructed for propagating information through the processing array.

In Chapter 4, we presented the basic mapping methodology used for implementing neural network computation. This mapping concept is based on neuron level parallelism, where one neuron is mapped to each of the processors in the processing array. The distributed operations associated with neural computation are performed by constructing computational paths which traverse all the necessary processors required for implementation of each specific distributed operation. In Chapter 5, we demonstrated how such computational paths can be constructed by embedding a processing ring of arbitrary length on the 2-D topology of the DREAM Machine. We have also shown how sparse but regularly connected neural network structures can be mapped onto the DREAM Machine by using several concurrently executing processing rings.

The communication flexibility attained by programmable switches in the DREAM Machine architecture, was shown to be useful in constructing variable-length processing rings. Being able to construct several variable-length processing rings , allows for efficient implementation of neural networks with a variety of interconnection structures. Even though this mapping approach is applicable to a wide range of structures, it can be inefficient in implementing sparse and irregularly connected neural network structures.

In Chapter 6, we presented the mapping problem as a conglomeration of three interdependent problems: the clustering problem, the assignment problem, and the scheduling problem. Although the primary objective of our work in Chapter 6 was to arrive at a method for efficient mapping of neural computation onto parallel processing architectures, the method and some of the cost functions associated with it were formalized such that they would be applicable to other problems in mapping parallel algorithms onto parallel processors.

The major contribution of Chapter 6 was to derive effective cost functions which evaluates the *goodness* of a specific assignment and scheduling for a given neural

network structure . We showed, using statistical physics techniques, how neural computation can be employed to find solutions to these problems by optimizing these cost functions. In Chapter 7, we presented the results of our optimization procedure in mapping several small example problems onto different parallel processing architectures. By comparing our results to randomly generated solutions, we presented empirical evidence indicating that our optimization routine generates efficient solutions. Furthermore, we demonstrated that our assignment optimization procedure has been able to consistently find better solutions than those found by a previously proposed optimization routine on a benchmark example [7].

In Chapter 8, we analyzed the asymptotic computational complexity of our implementation method and also that of the optimization procedures. We compared the results of our implementation method with those of others and showed that both results are equivalent in asymptotic terms. However, this equivalence holds only if no limits are placed on the interconnection network structure of the neural network models. Since realistic neural network models do not require full interconnection among all the neurons in the network, as evident by examining biological [3] as well as artificial neural networks applied to real-world problems [45], it can be shown that the implementation method described in this dissertation offers the best overall performance.

As neural network models evolve to be more complex, and as their sizes grow to address larger, more sophisticated problems, the need for efficient parallel implementations becomes ever more pressing. The DREAM Machine architecture, presented in this dissertation, is intended to be a special medium to address the current demands of neural network computation, and at the same time, offer sufficient flexibility for efficiently implementing future generations of neural network models.

9.2 Future Research Directions

The research presented in this dissertation encompasses a number of novel and promising areas in the design of parallel processing architectures and mapping methods for their efficient utilization. However, due to the wide scope of materials described in this dissertation, and the novelty of the concepts, much work remains to be done. The

future directions for this research can be divided into three separate areas as described below.

- The DREAM Machine architecture: The architectural description of the DREAM Machine, presented in Chapter 3, is given at a high level in order to demonstrate the specific features of the architecture used for efficient processing of neural networks. A more rigorous and detailed treatment of the architectural design is required to determine such details as the specific functional units, the data width, memory size, etc. to be used for an actual realization of this machine. Such detail design specifications must consider specific technologically imposed limitations of available hardware and their associated cost.

  Another possible direction for further exploration involves determining other applications where the DREAM Machine can be used for efficient processing. The flexibility in interprocessor communications, offered by this architecture, along with its indirect addressing capabilities, can be effectively utilized by a number of other applications in addition to neural networks.

- Mapping neural networks onto the DREAM Machine: Two methods for mapping neural network models onto the DREAM Machine where presented in this dissertation. The algorithmic mapping, described in Chapter 5, involves construction of variable-length rings on the DREAM Machine architecture. As eluded to in Chapter 5, the determination of optimum ring lengths for sparse but regularly connected neural networks is complex and requires an optimization procedure. In addition to further research in developing such an optimization tool, an automatic method for physically embedding variable-length rings on the 2-D lattice of the DREAM Machine should b devised in order to simplify the mapping process.

  In the area of optimization based mapping method for implementing neural networks, the approach described in this dissertation should be extended to efficiently address multilayer neural networks. Also, concurrent implementation of the assignment and scheduling optimization routines should be studies in detail. This approach has a potential for arriving at good solutions for the scheduling routine by allowing for interaction between both optimization procedures.

Implementation of such an approach requires analysis of the convergence characteristic of the system having interdependent variables.

- Automatic mapping of parallel algorithms onto parallel architectures using optimization techniques: As mentioned in Chapter 6, the optimization based mapping, proposed in this dissertation, can be applied to a variety of similar problems in mapping of parallel algorithms onto parallel processing architectures. Further research is required, particularly in the scheduling optimization area, for demonstrating the effectiveness of this method for a wider range of problems in parallel processing. Currently, work is in progress in applying this technique to the sparse matrix bandwidth reduction problem and efficient implementation of matrix-vector operations on systolic arrays [74]. Research into efficient mappings of data-flow graphs onto parallel architectures is also a promising and natural direction to follow.

References

[1] DARPA Neural Network Study. AFCEA International Press, 1988.

[2] B. Angeniol, G. D. L. C. Vaubois and J.-Y. L. Texier, "Self-Organizing Feature Maps and the Traveling Salesman Problem." *Neural Networks*. **1**: 289-293, 1988.

[3] M. A. Arbib, The Metaphorical Brain 2, Neural Networks and Beyond. John Wiley & Sons, 1989.

[4] B. A. Armstrong, "A Hybrid Algorithm for Reducing Matrix Bandwidth." *Intr. Jour. for Num. Meth. in Engr.* **20**: 1929-1940, 1984.

[5] G. Blelloch and C. R. Rosenberg, "Network Learning on the Connection Machine," *Proceedings of the 10th Intern. Joint Conf. on Artificial Intelligence*, Milan, Italy, pp. 323-326, 1987.

[6] D. W. Blevins, E. W. Davis, R. A. Heaton and J. H. Reif, "BLITZEN: A Highly Integrated Massively Parallel Machine." *Journal of Parallel and Distributed Computing*. **8**: 150-160, 1990.

[7] S. H. Bokhari, "On the Mapping Problem." *IEEE Transactions on Computers*. **C-30**(3): 207-214, 1981.

[8] S. H. Bokhari, Assignment Problems in Parallel and Distributed Computing. Kluwer Academic Publishers, 1987.

[9] J. R. Brown, M. M. Garber and S. F. Venable, "Artificial Neural Network on a SIMD Architecture," *Proceedings of the Symposium on the Frontiers of Massively Parallel Computations*, pp. 43-47, 1988.

[10] G. A. Carpenter and S. Grossberg, "A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine." *Compute Vision, Graphics, and Image Processing*. **37**: 54-115, 1987.

[11] A. J. De Groot and S. R. Parker, "Systolic Implementation of Neural Networks," *Proceedings of the SPIE Vol. 1058 High Speed Computing II*, Los Angeles, Ed. K. Bromley, pp. 182-190, 1989.

[12] E. Deprit, "Implementing Recurrent Back-Propagation on the Connection Machine." *Neural Networks*. **2**: 295-314, 1989.

[13] R. Durbin and D. Willshaw, "An Analogue Approach to the Travelling Salesman Problem Using an Elastic Net Method." *Nature*. **326**: 689-691, 1987.

[14] G. C. Everstine, "A Comparison of Three Resequencing Algorithms for the Reduction of Matrix Profile and Wavefront." *International Journal of Numerical Methods in Engineering*. **14**: 837-853, 1979.

[15] L. Fang and T. Li, "Design of Competiotion-Based Neural Networks for Combinatorial Optimization." *International Journal of Neural Systems*. **1**(3): 221-235, 1990.

[16] C. T. Fike, Computer Evaluation of Mathematical Functions. Prentice-Hall, 1968.

[17] M. J. Flynn, "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers*. **21**: 948-960, 1972.

[18] B. M. Forrest, D. Roweth, N. Stroud, D. J. Wallace and G. V. Wilson, "Implementing Neural Network Models on Parallel Computers." *The Computer Journal*. **30**(5): 413-419, 1987.

[19] K. Fukushima, "Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition." *Neural Networks*. **1**: 119-130, 1988.

[20] J.-L. Gaudiot, C. v. d. Malsburg and S. Shams, "A Data-Flow Implementation of a Neurocomputer for Pattern Recognition Applications," *Proceedings of the Fourth Annual Aerospace Applications of Artificial Intelligence*, Dayton, OH, Vol. 1, pp. 327-338, 1988.

[21] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Publishing Co., 1989.

[22] H. P. Graf, L. D. Jackel and W. E. Hubbard, "VLSI Implementation of a Neural Network Model." *Computer*. **21**(3): 41-49, 1988.

[23] K. A. Grajski, "Neurocomputing Using the MasPar MP-1," Technical Report No. 90-010, Ford Aerospace Corp., Oct. 1, 1990

[24] M. K. Habib and H. Akel, "A Digital Neuron-Type Processor and Its VLSI Design." *IEEE Transactions on Circuits and Systems*. **36**(5): 739-746, 1989.

[25] D. Hammerstorm, "A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning," *Proceedings of the Inter. Joint Conf. on Neural Networks*, San Diego, Vol. 2, pp. 537-544, 1990.

[26] R. Hecht-Nielsen, Neurocomputing. Addison-Wesley Publishing, 1990.

[27] W. D. Hillis, The Connection Machine. Cambridge MA, MIT Press, 1985.

[28] G. Hinton, D. Ackley and T. Sejnowski, "Boltzmann Machines: Constraint Satisfaction networks that learn," Technical Report CMU-CS-84-119, Carnegie-Mellon University, Dept. of Computer science,

[29] Y. Hirai, K. Kamada, M. Yamada and M. Ooyama, "A Digital Neuro-Chip with Unlimited Connectability for Large Scale Neural Networks," *Proceedings of the Inter. Joint Conf. on Neural Networks*, Washington D.C., Vol. 2, pp. 163-169, 1989.

[30] A. Hiraiwa, M. Fujita, S. Kurosu, S. Arisawa and M. Inoue, "Implementation of ANN on RISC Processor Array," *Proceedings of the Inter. Conf. on Appl. Spec. Array Proc.*, Princeton, NJ, Ed. S. Y. Kung, E. E. Swartzlander, J. A. B. Fortes and K. W. Przytula, pp. 677-687, 1990.

[31] M. Holler, S. Tam, H. Castro and R. Benson, "An Electrically Trainable Artificial Neural Network (ETANN) with 10240 "Floating Gate" Synapses," *Proceedings of the Inter. Joint Conf. on Neural Networks*, Washington D.C., Vol. 2, pp. 191-196, 1989.

[32] J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities." *Proceedings of the National Academy of Science USA*. **79**: 2554-2558, 1982.

[33] J. J. Hopfield and D. W. Tank, ""Neural" Computation of Decisions in Optimization Problems." *Biological Cybernetics*. **52**: 141-152, 1985.

[34] A. Iwata, Y. Yoshida, S. Matsuda, Y. Sato and N. Suzumura, "An Artificial Neural Network Accelerator Using General Purpose 24 bits Floating Point Digital Signal Processors," *Proceedings of the Inter. Joint Conf. on Neural Networks*, Washington D.C., Vol. 2, pp. 171-175, 1989.

[35] F. A. Kamangar, R. A. Duderstadt and J. O. Smith, "Efficient Implementation of Connectionist Models on MIMD Parallel Processors Using Chordal Ring Topologies," *Proceedings of the Inter. Joint Conf. on Neural Networks*, Washington D.C., Vol. 2, pp. 588, 1989.

[36] S. Kirkpatrick, C. D. Gelatt Jr and M. P. Vecchi, "Optimization by Simulated Annealing." *Science*. **220**: 671-680, 1983.

[37] T. Kohonen, Self-Organization and Associative Memory. second ed., Springer Series in Information Sciences. Springer-Verlag, 1987.

[38] B. Kosko, "Bidirectional Associative Memories." *IEEE Transactions on Systems, Man, and Cybernetics*. **18**: 49-60, 1988.

[39] B. Kosko, "Unsupervised Learning in Noise." *IEEE Transactions on Neural Networks*. **1**(1): 44-57, 1990.

[40] H. T. Kung and C. E. Leiserson. "Systolic Arrays for VLSI." In Introduction to VLSI Systems. Ed. C. Mead and L. Conway, Reading, MA, Addison-Wesley, pp. 1980.

[41] S. Y. Kung and J. N. Hwang, "Systolic Architectures for Artificial Neural Nets," *Proceedings of the IEEE Inter. Conf. on Neural Networks*, San Diego, CA, Vol. 2, pp. 165-172, 1988.

[42] S. Y. Kung and J. N. Hwang, "A Unified Systolic Architecture for Artificial Neural Networks." *Journal of Parallel and Distributed Computing*. **6**: 358-387, 1989.

[43] H. K. Kwan and P. C. Tsang, "Systolic Implementation of Multi-Layer Feed-Forward Neural Network with Back-Propagation Learning Scheme," *Proceedings of the Inter. Joint Conf. on Neural Networks*, Washington D.C., Vol. 2, pp. 155-158, 1990.

[44] E. L. Lawler, J. K. Lenstra, A. H. Kan, G. Rinnooy and D. B. Shmoys. The Traveling Salesman Problem: A Gauided Tour of Combinatorial Optimization, Wiley, New York, 1985.

[45] Y. Le Cun, B. Bowser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel. "Handwritten Digit Recognition with a Back-Propagation Network." In Advances in Neural Information Processing Systems 2. Ed. D. S. Touretzky, San Mateo, CA, Morgan Kaufmann, pp. 396-403, 1990.

[46] Y. Le Cun, J. S. Denker and S. A. Solla. "Optimal Brain Damage." In Advances in Neural Information Processing Systems 2. Ed. D. S. Touretzky, San Mateo, CA, Morgan Kaufmann, pp. 598-605, 1990.

[47] W.-M. Lin, V. K. Prasanna and K. W. Przytula, "Algorithmic Mapping of Neural Network Models onto Parallel SIMD Machines." *IEEE Transactions on Computers*. **40**(12): 1390-1401, 1991.

[48] M. J. Little and J. Grinberg. "The 3-D Computer: An Integrated Stack of WSI Wafers." In Wafer Scale Integration. Ed. E. Swartrzlander, Boston, Kluwer, pp. Chapter 8, 1988.

[49] R. Mann and S. Haykin, "A Parallel Implementation of Kohonen Feature Maps on the Warp Systolic Computer," *Proceedings of the Inter. Joint Conf. on Neural Networks*, Washington D.C., Vol. 2, pp. 84-87, 1990.

[50] V. Milutinovic, "Mapping of Neural Networks on the Honeycomb Architecture." *Proceedings of the IEEE*. **77**(12): 1875-1878, 1989.

[51] M. Misra and V. K. Prasanna Kumar, "Massive Memory Organization for Implementing Neural Networks," *Proceedings of the Inter. Conf. on Pattern Recognition*, Vol. 2, pp. 259-264, 1990.

[52] M. Misra and V. K. Prasanna Kumar, "Neural Network Simulation on a Reduced Mesh of Trees Organization," *Proceedings of the SPIE/SPSE Symp. on Electronic Imaging*, 1990.

[53] A. Moopenn, T. Duong and A. P. Thakoor, "Digital-Analog Hybrid Synapse Chips for Electronic Neural Networks," *Proceedings of the Advances in Neual Information Processing Systems 2*, Denver, CO, Ed. D. S. Touretzky, pp. 769-776, 1989.

[54] N. Morgan. Artificial Neural Networks: Electronic Implementations, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[55] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman and J. Beer, "The RAP: A Ring Array Processor for Layered Network Calculations," *Proceedings of the Inter. Conf. on Appl. Spec. Array Proc.*, Princeton, NJ, Ed. S. Y. Kung, E. E. Swartzlander, J. A. B. Fortes and K. W. Przytula, pp. 296-308, 1990.

[56] A. Namphol, M. Arozullah and S. Chin, "Higher Order Data Compression With Neural Networks," *Proceedings of the Inter. Joint Conf. on Neural Networks*, Seattle, WA, Vol. I, pp. 55-59, 1991.

[57] D. Parkinson and J. Litt. Massively Parallel Computing with the DAP, MIT Press, Cambridge, Massachusetts, 1989.

[58] C. Peterson and B. Soderberg, "A New Method for Mapping Optimization Problems Onto Neural Networks." *International Journal of Neural Systems*. **1**(1): 3-22, 1989.

[59] F. Piazza, M. Marchesi, G. Orlandi and A. Unicini, "Coarse-Grained Processor Array Implementing the Multilayer Neural Network Model," *Proceedings of the Inter. Symp. on Cir. & Sys.*, New Orleans, LA, Vol. 4, pp. 2963-2966, 1990.

[60] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky and H. T. Kung, "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second," *Proceedings of the IEEE International Confer. on Neural Networks*, San Diego, 1988.

[61] V. K. Prasanna Kumar and K. W. Przytula, "Algorithmic Mapping of Neural Network Models onto Parallel SIMD Machines," *Proceedings of the Inter. Conf. on Appl. Spec. Array Proc.*, Princeton, NJ, Ed. S. Y. Kung, E. E. Swartzlander, J. A. B. Fortes and K. W. Przytula, 1990.

[62] K. W. Przytula, "Systolic/Cellular System," Internal Report, Hughes Research Labs., Aug. 1989

[63] K. W. Przytula and J. G. Nash, "A Special Purpose Coprocessor for Signal Processing," *Proceedings of the 21st Asilomar Conference on Signals, Systems and Computers*, Monterey, CA, pp. 736-740, 1987.

[64] U. Ramacher and U. Ruckert. VLSI Design of Neural Networks, Kluwer Academic Publishers, 1991.

[65] F. Rosenblatt, "The Perceptron; A Probabilistic model for information storage and organization in the brain." *Psychology Review*. **65**: 386-408, 1958.

[66] D. E. Rumelhart, G. E. Hinton and R. J. Williams. "Learning Internal Representations by Error Propagation." In Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1, Ed. D. E. Rumelhart and J. McClelland, Cambridge, MIT Press, pp. 318-364, 1986.

[67] S. Satyanarayana, Y. Tsividis and H. P. Graf. "A Reconfigurable Analog VLSI Neural Network Chip." In Advances in Neural Information Processing Systems 2. Ed. D. Touretzky, San Mateo, CA, Morgan Kaufmann, pp. 758-768, 1990.

[68] D. B. Schwartz, R. E. Howard and W. E. Hubbard, "A Programmable Analog Neural Network Chip." *IEEE Journal of Solid-State Circuits*. **24**(2): 313-319, 1989.

[69] T. J. Sejnowski and C. R. Rosenberg, "Parallel Networks that Learn to Pronounce English Text." *Complex Systems*. **1**: 145-168, 1987.

[70] S. Shams, "Neural Networks for Passive Sonar Target Detection," Technical Report, Hughes Research Labs, Dec. 1991

[71] S. Shams and J.-L. Gaudiot, "Efficient Implementation of Neural Networks on the DREAM Machine," *Proceedings of the 11th Inter. Conf. on Pattern Recognition*, The Hague, The Netherlands, 1992.

[72] S. Shams and K. W. Przytula, "Mapping of Neural Networks onto Programmable Parallel Machines," *Proceedings of the Intern. Symp. on Circuits and Systems*, New Orleans, LA, Vol. 4, pp. 2613-2617, 1990.

[73] S. Shams and K. W. Przytula. "Implementation of Multilayer Neural Networks on Parallel Programmable Digital Computers." In Parallel Algorithms and Architectures for DSP Applications. Ed. M. Bayoumi, Kluwer Acadamic Publishers, pp. 225-253, 1991.

[74] S. Shams and P. Simic, "Efficient Mapping of Sparse Iterative Systems onto Parallel Systolic Architectures Using Constrained Nets," submitted for publication in the *Proceedings of the Neural Information Processing Systems 92*.

[75] D. B. Shu, J. G. Nash, M. M. Eshaghian and K. Kim. "Implementation and Application of a Gated-connection Network in Image Understanding." In Reconfigurable Massively Parallel Computers. Ed. H. Li and Q. F. Stout, Prentice Hall, 1991.

[76] P. Simic and S. Shams, "Solving the Assignment and Scheduling Problems Using Cnet," Technical Report CALT-68-1892, California Institute of Technology, 1992.

[77] P. D. Simic, "Statistical Mechanics as the Underlying Theory of 'Elastic' and 'Neural' Optimisations." *Networks*. **1**: 89-103, 1990.

[78] P. D. Simic, "Constrained Nets for Graph Matching and Other Quadratic Assignment Problems." *Neural Computation*. **3**: 268-281, 1991.

[79] A. Singer, "Implementatins of Artificial Neural Networks on the Connection Machine." *Parallel Computing*. **14**(3): 305-315, 1990.

[80] D. F. Specht, "Probabilistic Neural Network." *Neural Networks*. **3**: 109-118, 1990.

[81] H. Szu, "Fast Simulated Annealing," *Proceedings of the AIP Conference Proceedings 151: Neural Networks for Computing*, New York, Ed. J. Denker, pp. 420-425, 1986.

[82] H. H. Thodberg, "Improving Generalization of Neural Networks Through Pruning." *International Journal of Neural Systems*. **1**(4): 317-326, 1991.

[83] S. T. Toborg and K. Hwang, "Cooperative Vision Integeration Through Data-Parallel Neural Computations." *IEEE Transactions on Computer*. **40**(12): 1368-1379, 1991.

[84] S. Tomboulian. "Introduction to a System for Implementing Neural Net Connections on SIMD Architectures." In Neural Information Processing Systems. Ed. D. Z. Anderson, New York, American Institute of Physics, pp. 804-813, 1988.

[85] J. D. Ullman, Computational Aspects of VLSI. Computer Science Press, 1984.

[86] C. C. Weems. (1984). Image Processing on a Content Addressable Array Parallel Processor, PhD Dissertation, University of Massachusetts, Amherst, MA.

[87] S. S. Wilson, "Neural Computing on a One Dimensional SIMD Array," *Proceedings of the Inter. Joint Conf. on Arti. Intel.*, pp. 206-211, 1989.

[88] M. Witbrock and M. Zagha, "An Implementation of Back-Propagation Learning on GF11, a Large SIMD Parallel Computer." *Parallel Computing*. **14**(3): 329-346, 1990.

[89] X. Zhang, M. McKenna, J. P. Mesirov and D. L. Waltz. "An Efficient Implementation of the Back-Propagation Algorithm on the Connection Machine CM-2." In Advances in Neural Information Processing Systems 2. Ed. D. S. Touretzky, San Mateo, CA, Morgan Kaufmann, pp. 801-809, 1990.