1997

Efficient Mapping of Neural Network Models on a Class of Parallel Architectures.

Behnam Seyed Arad
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

EFFICIENT MAPPING OF NEURAL NETWORK MODELS ON A CLASS OF PARALLEL ARCHITECTURES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by

Behnam Seyed Arad

B.S., University of Massachusetts, Lowell, 1988
M.S., Purdue University, 1990
May 1997

ACKNOWLEDGMENTS

I would like to thank God, most gracious, most merciful for guiding me throughout my life and providing me with the opportunity and patience to complete this project. I would like to express my sincere appreciation to my major advisor, Dr. Ahmed El-Amawy, for providing me with the opportunity to intellectually challenge myself as an engineer. His advice as a mentor is highly valued and will serve me well in the future. I would like to thank him for the support and guidance he provided me throughout my studies at L.S.U. and the opportunity to benefit from financial support through his grant.

Sincere gratitude is extended to Dr. Alexander Skavantzos and Dr. Ramachandran Vaidyanathan for their invaluable advice and insight into my academic pursuits both here at L.S.U. and in future endeavors. My thanks to Dr. Gil S. Lee, Dr. John Tyler, and Dr. Tryfon T. Charalampopoulos for serving as members of my graduate committee.

I wish to thank my parents for their love, patience, support and encouragement throughout my life. Had it not been for them, I could never have found the strength, determination, and fortitude necessary to pursue this degree. I would also like to extend my gratitude to my in-laws Mr. and Mrs. Jazbi for their support and understanding throughout this work. A special acknowledgment to my late grandmother, Bebe Sakineh, for her love and kindness. My special thanks are extended to my dear friends Dr. Ramezanian, Dr. Sabahi, and Ali Zandieh for their sincere friendship and support over the years.

Finally, I would like to thank the love of my life, Parisa, for her support, patience, and encouragement throughout the difficult stages of this project. I can never thank her for the difficulties she had to put up with.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This dissertation develops a formal and systematic methodology for efficient mapping of several contemporary artificial neural network (*ANN*) models on k-ary n-cube parallel architectures (*KNC's*). We apply the general mapping to several important *ANN* models including feedforward *ANN*'s trained with backpropagation algorithm, radial basis function networks, cascade correlation learning, and adaptive resonance theory networks.

Our approach utilizes a parallel task graph representing concurrent operations of the *ANN* model during training. The mapping of the *ANN* is performed in two steps. First, the parallel task graph of the *ANN* is mapped to a virtual *KNC* of compatible dimensionality. This involves decomposing each operation into its atomic tasks. Second, the dimensionality of the virtual *KNC* architecture is recursively reduced through a sequence of transformations until a desired metric is optimized. We refer to this process as folding the virtual architecture. The optimization criteria we consider in this dissertation are defined in terms of the iteration time of the algorithm on the folded architecture. If necessary, the mapping scheme may utilize a subset of the processors of a given *KNC* architecture if it results in the most efficient simulation. A unique feature of our mapping is that it systematically selects an appropriate degree of parallelism leading to a highly efficient realization of the *ANN* model on *KNC* architectures.

A novel feature of our work is its ability to efficiently map unit-allocating *ANN*'s. These networks possess a dynamic structure which grows during training. We present a highly efficient scheme for simulating such networks on existing *KNC* parallel architectures. We assume an upper bound on size of the neural network. We perform the folding such that

the iteration time of the largest network is minimized. We show that our mapping leads to near-optimal simulation of smaller instances of the neural network . In addition, based on our mapping no data migration or task rescheduling is needed as the size of network grows.

CHAPTER 1

INTRODUCTION

1.1 Artificial Neural Networks

The struggle to understand the human nervous system led to the discovery of neurons as structural components of the brain. A typical neuron is five to six orders of magnitude slower than typical electronic gates [35]. However, the slow processing rate of neurons is compensated by the large number of neurons and their dense connectivity in the brain. Shepherd and Koch [72] estimated the number of neurons in human cortex to be 10 billion, with 60 trillion connections among them. This makes human brain an extremely efficient system [35].

Artificial Neural Networks (*ANN*'s) also known as neuro-computers, connectionist networks, and parallel distributed processors [35] are biologically motivated systems which represent simulated models of a real nervous system. They comprise of densely interconnected processing units called *artificial neurons*. *ANN*'s are capable of storing knowledge and making it available for use [35]. The procedure used to store knowledge in an *ANN* is called *learning*. Learning constitutes an important feature of *ANN*'s. Unlike conventional computers which tackle problems through programming, *ANN*'s are capable of solving problems through learning.

ANN's are viable computational models for tackling complex and large scale problems intractable by conventional computers. These models are suitable for applications where explicit knowledge is not available [71]. In particular, *ANN*'s can be applied to problems which are difficult or impossible to express mathematically [71]. *ANN*'s have been

applied to a wide variety of problems including pattern classification, speech synthesis and recognition, adaptive interfaces between human and complex physical systems, function approximation, image compression, associate memory, clustering, forecasting and prediction, combinatorial optimization, nonlinear system modeling, and control [33]. The computing power of *ANN*'s can be attributed to their massively parallel distributed architecture and to their ability to learn and generalize [35].

ANN's are made up of artificial neurons which are connected according to various architectures. Links between neurons are assigned weights to resemble biological synapses. By properly adjusting its weights and transition function, an *ANN* can realize a relation between an input space and an output space [62]. The realized relation depends on several factors including the network structure, threshold and weight values, and the nature of network's dynamics [62]. Next we briefly review several important aspects of *ANN*'s.

1.1.1 Artificial Neuron

Figure 1.1 shows a simplified biological neuron. The human brain contains approximately 10 billion such cells, each being connected to about 10^4 other cells [62]. The actual operation of a nerve cell is a mystery. However, scientists have been able to reasonably approximate how a nerve cell operates. The operation of a cell can be stated as follows. Each cell receives (electrochemical) signals through its input branches called *dendrites*, which accept outputs of adjacent cells. Each input signal can be excitatory (positive value) or inhibitory (negative value). If the overall excitatory input signals of a cell are strong enough (exceeding a threshold), the cell will transmit an electrical pulse along

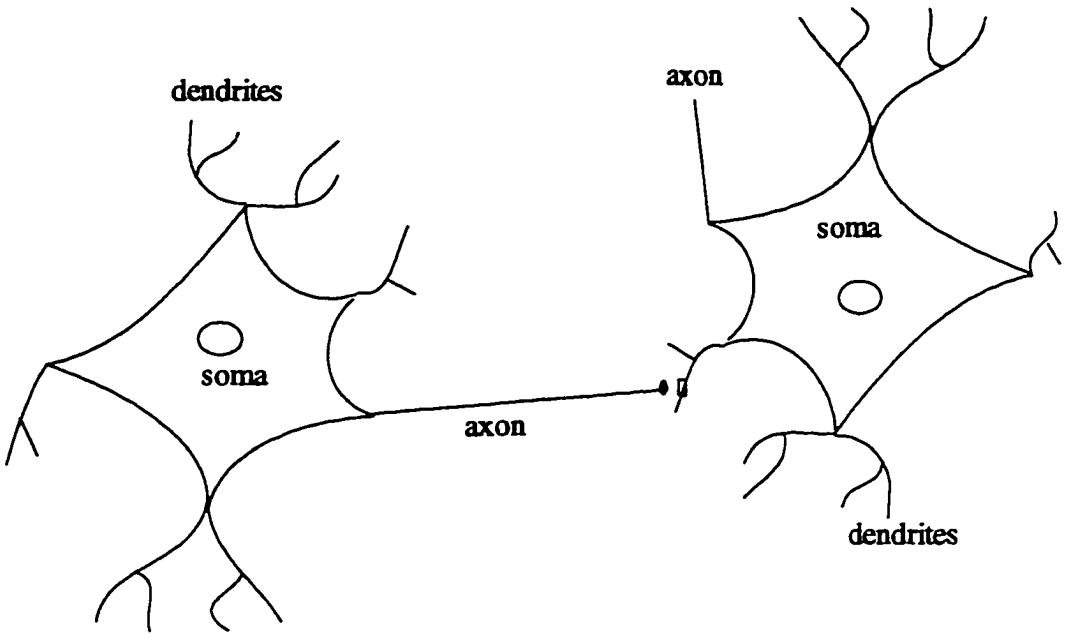

Figure 1.1: A simplified biological neuron.

its output branch, called *axon*, to the dendrites of its neighboring cells. The junction between the axon of one cell and the dendrite of another cell is called *synapse*.

The basic component of an *ANN* is an artificial neuron. It is an abstract mathematical model developed to mimic the behavior of a nerve cell. McCulloch and Pitts [52] modeled the biological nerve cell using a binary threshold unit [62]. Their model is shown in Figure 1.2. It has n inputs ($x_1, x_2,...,x_n$) resembling the dendrites of a nerve cell. A weight (w_i) is associated with each input (x_i) to indicate the strength of the connection between the cell and a neighboring neuron. These are called synaptic weights. The model computes the sum of products of its inputs and their corresponding weights. If the computed sum is above a certain threshold (θ), the neuron's output (y) is set to 1. Otherwise, the output remains at zero.

The McCulloch and Pitts model can be generalized to allow various transition functions. The *transition function* determines the output of an artificial neuron based on

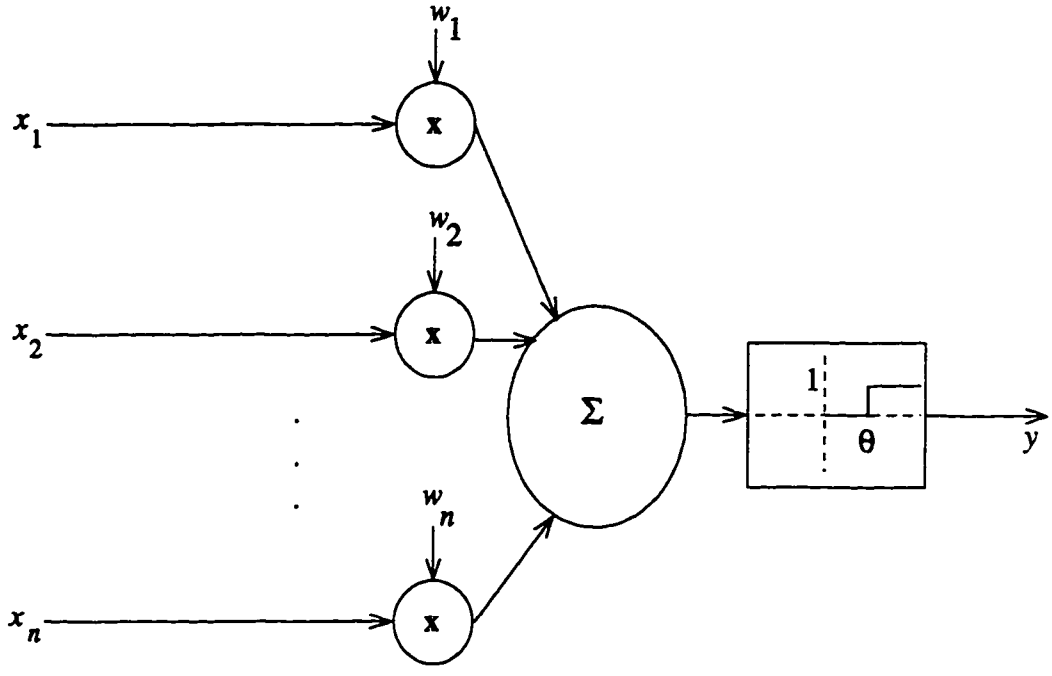

Figure 1.2: The McCulloch and Pitts model.

the weighted sum of its inputs. The artificial neuron model used in this dissertation is depicted in Figure 1.3. Typical transition functions are the identity, threshold, piecewise linear, and sigmoid. These functions are shown in Figure 1.4. We will refer to an artificial neuron as simply *neuron* or *unit* henceforth.

1.1.2 Structure of *ANN*'s

ANN models are structured in various ways. The structure is generally linked to the scheme used to make the *ANN* realize an input to output relation [35]. There are different classifications of *ANN* structures. Here, we consider the four classes identified in [35], namely *single-layer feedforward*, *multilayer feedforward*, *recurrent*, and *lattice structure*. An example of each class is depicted in Figure 1.5. The simplest structure is a single layer of neurons. This network simply projects inputs to a layer of neurons referred to as the output layer. Basic *associate neural memory* [33] is an example of this class.

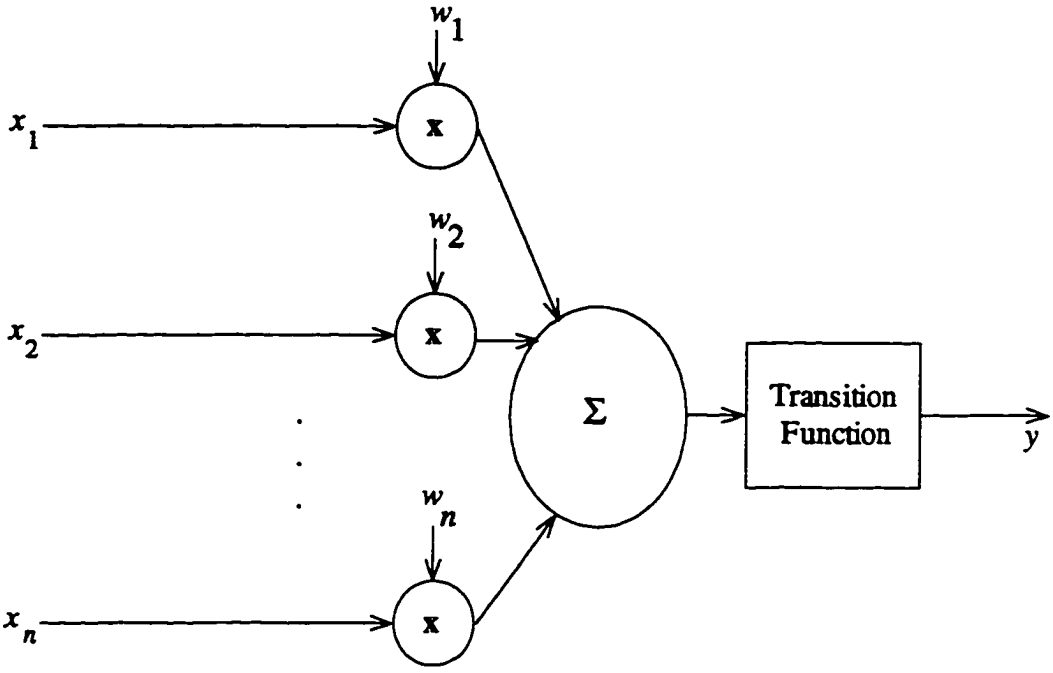

Figure 1.3: An artificial neuron.

A multilayer feedforward network comprises of more than one layer of neurons. Neuron layers other than the input and output layers are referred to as *hidden layers*, and their corresponding units are called *hidden neurons*. Nodes in each layer are connected to nodes in adjacent layers. In certain models, layers are fully connected, i.e., each neuron is fully connected to all neurons in adjacent layers. This class includes a very popular network called *feedforward neural network* which is typically trained with the *backpropagation learning* [69] algorithm. Networks may also be partially connected, i.e., a neuron is connected to a subset of neurons in adjacent layers. The *locally connected network* [35] is an example of such structures . A lattice network is very similar to a feedforward network except that output neurons are arranged in arrays (rows or columns).

As opposed to feedforward networks, the recurrent network includes at least one feedback link. A *feedback link* connects the output of a neuron to inputs of neuron(s) in previous layers. The most famous recurrent network is the *Hopefield network* [62]. It

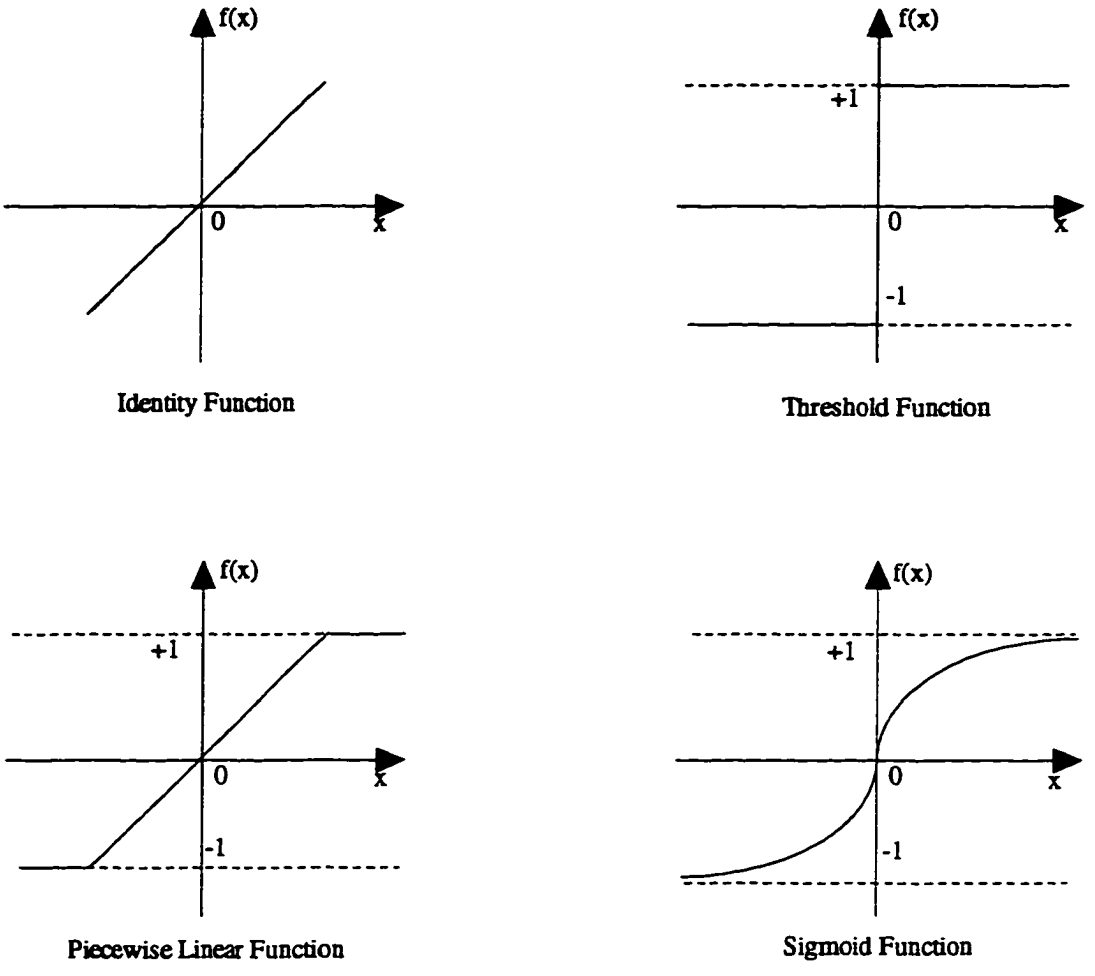

Figure 1.4: Common transition functions.

consists of a single layer of neurons. The output of each neuron is connected to inputs of other neurons. The feedback loop has a significant impact on the storage and learning capability of the network [35].

1.1.3 *ANN*'s Learning (Training) Process

One of the most profound features of *ANN*'s is their ability to learn. Conventional digital computers are generally programmed (hardware or software) to perform a certain task. *ANN*'s on the other hand are trained by working with examples rather that algorithms [71]. Formally, learning constitutes incrementally adjusting weights of the network so that

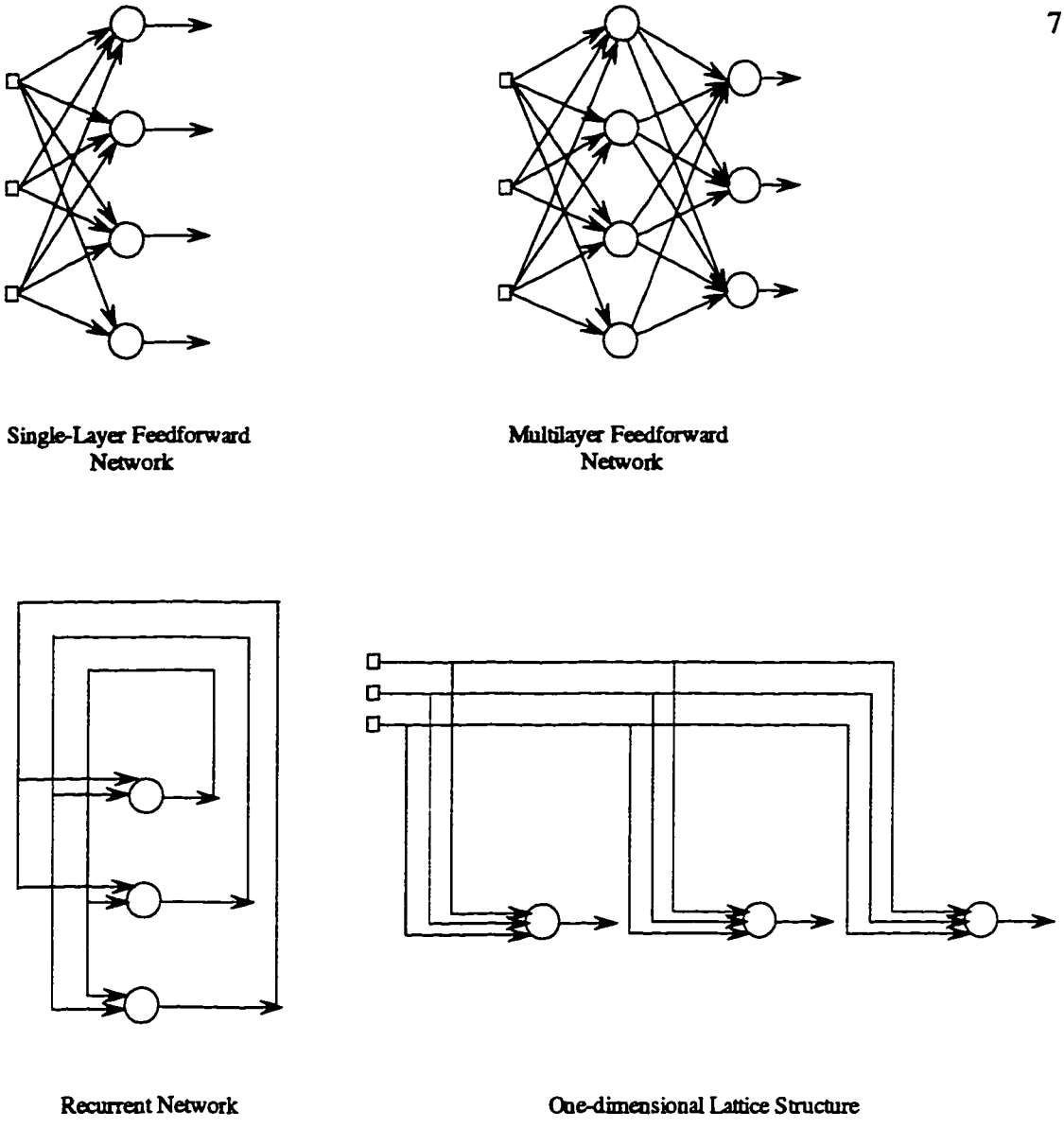

Figure 1.5: Neural network structures.

it can realize an input to output relation. The learning generally involves improving a predefined performance measure over time [33]. Hence, learning can be viewed as an optimization search [33]. The scheme followed to train a network is referred to as the *learning* (*training*) algorithm.

Learning algorithms are classified into three major categories, *supervised*, *unsupervised*, and *reinforced* training [33]. In supervised training a desired output is

associated with each input pattern [33]. Generally, the algorithm is designed to reduce the error between the desired and actual output patterns. Error backpropagation learning [69] and *cascade correlation* [25] employ supervised training. Unsupervised training, on the other hand classifies the input patterns into different categories through optimization of some criterion [33]. *Hebbian learning* [36], *Adaptive Resonance Theory (ART1)* [31], and *concept-forming cognitive* models [3] are examples of unsupervised training. Reinforcement learning constitutes updating the weights in order to maximize the probability of a factor called *reinforcement signal* [33]. This scheme originated in connection with experimental studies of animal learning [33]. *Associate Reward-Penalty Reinforcement rule* [7] is an example of reinforcement learning.

1.2 Implementation of *ANN*'s

Neural processing can gain acceptance as a practical problem solving tool only if it provides significant performance improvement over conventional methods. In principle, neural processing may be the only viable tool for solving many difficult scientific and engineering problems. However, only cost-effective implementations of *ANN* models can lead to their practicality. The problem of efficient realization of *ANN*'s has been under study since the late 1980's. The studies include both hardware and software approaches.

ANN's are extremely computationally demanding. In particular, *ANN* training algorithms are computationally intensive procedures and are generally rather slow in nature. Most training algorithms require large number of iterations and hence time to converge. Software and hardware implementations have been proposed in an attempt to efficiently

realize these models and enhance their practicality. Recent advances in simulation methods and neurohardware have made *ANN*'s more practical [71].

Hardware implementations have utilized digital and analog technologies. Examples of hardware implementations include: a low-power VLSI arrhythmia classifier [48], fully parallel stochastic neural networks [79], optoelectronic VLSI shunting neural network [39], a single chip realizing 10^6-synapse neural network [84], a generic systolic array building block for on-chip learning [47], the pRAM chip [21], an analog CMOS chip set for neural networks with arbitrary topology [46], a general purpose neurochip [23], a CMOS implementation of neural network models [30], a VLSI architecture for on-chip learning [32], a neuromorphic VLSI learning system [2], and visual computations using analog CMOS processing arrays [74].

Digital approaches (in particular those based on VLSI technology) are considered well suited for most neural network applications [35][43]. The principle reasons for utilizing VLSI technology are: 1) functional density available on VLSI chips, 2) many *ANN*'s have regular topology, 3) *ANN*'s require a few simple and well defined arithmetic operations [11], and 4) the ease and low cost of mass production of *VLSI* chips. Hitachi's WSI chip with 576 digital neurons and 36k weights integrated onto a 5-inch silicon wafer using 0.8μ CMOS technology is one of the fastest neural chips [71]. A system built based on eight such WSI boards produces 2.3 giga connection updates per second (*CUPS*) for a backpropagation network [71].

Analog electronics on the other hand provide high speed [43][71], packing density, and low power consumption [71]. However, problems such as low precision [43], analog

storage of weights, susceptibility to temperature changes and interference [71] make analog neural networks less practical than digital approaches. Nevertheless, several analog neural chips have been developed including Mitsubishi's Neurochip [43]. It contains 336 neurons and 28k connections. The chip can provide 28 giga *CUPS* during the learning phase.

Despite advances in neurohardware technology most realizations of *ANN*'s are still done in software on general-purpose computers. The process of simulating *ANN*'s on conventional computers is referred to as *mapping*. The popularity of software simulations is due to several reasons. First, software simulations provide flexibility, which is a crucial criterion for experimental work in the field [49]. Through software implementations researchers have been able to explore different characteristics of *ANN* models. Such studies may not be as cost-effective if done on neurohardware. Second, neural processing on general purpose computers provides insight into the behavior of *ANN* models in an attempt to determine how these models can be efficiently implemented in hardware. For instance, neural processing on parallel computers has been essential in determining which parallelization techniques are most suitable for hardware, and which hardware feature can improve efficiency [71]. Third, hardware implementation of certain *ANN*'s is not feasible. For instance, the *Cascade Correlation* learning architecture [25] which consists of a cascaded network of neurons cannot be easily implemented using VLSI technology [64].

Conventional serial computers are generally too slow for computationally intensive *ANN* models, especially for large networks. In particular, processing demands of the learning algorithms of *ANN*'s are considerable. A common approach for efficient software realization of *ANN*'s is to map them onto parallel computers. Parallelism is at the very heart

of neural processing [71]. *ANN* models exhibit several characteristics which favor parallel implementation. These include large number of simple operations which can be performed concurrently, and distributed memory requirements.

The mapping techniques used for simulating *ANN*'s onto parallel architectures are categorized into two general groups: ***heuristic mapping*** and ***algorithmic mapping*** [49]. Heuristic mapping schemes rely on trial and error methods based on familiarity with *ANN* algorithms and the target machines [49]. Examples of heuristic approaches include the realization of multilayer perceptron on the MPP [34], Warp [67], Connection Machine [8], and Hughes Systolic Cellular/Processor [79]. Algorithmic mapping schemes on the other hand are systematic implementations of *ANN* models on target architectures. Examples of such mappings include implementation of feedforward *ANN* onto multiple bus [24], and hypercube based systems [51][42]. They also include simulation of *ANN's* on mesh-connected SIMD machines [49].

Several factors should be considered in any attempt to develop efficient mappings of *ANN*'s onto parallel architectures. These factors include specific features of the *ANN* model and characteristics of the target parallel architectures [71]. Processing and communication demands of the *ANN* should be studied at the initial stages of the mapping process. As mentioned earlier, *ANN*'s generally have high computational demands especially during the learning phase. In addition, the communication demands of *ANN*'s are also considerable due to their massive interconnectivity. For that reason, broadcast operations could be very efficient [24].

It is essential to explore parallelization strategies appropriate for a given *ANN* model. Parallelism can be exploited at different levels of *ANN* training. The degree of parallelism attained at each level is related to the granularity of tasks at that level. Several categorizations of mapping algorithms based on the amount of parallelism appear in the literature including the classifications in [40] and [59]. These categorizations are typically made with a specific *ANN* model in mind. However, to some extent they can also be applied to other models [71]. The classification due to Nordstorm and Svesson [71] includes the following parallelization approaches:

- *Session parallelism* (simultaneous execution of different sessions)
- *Training parallelism* (concurrent training using different samples [61])
- *Layer parallelism* (simultaneous execution of different network layers)
- *Node parallelism* (concurrent execution of node operations for a single input)
- *Weight parallelism* (concurrent weight summation per node)

Each approach utilizes the parallelism at a different level. It is crucial to determine which approach is suitable for a given *ANN* model. The characteristics of the target architecture and the limitations it might put on problem decomposition should also be considered.

As far as mapping is concerned, parallel processing machines can be divided into two broad categories: *single instruction stream multiple data stream* (SIMD) and *multiple instruction stream multiple data stream* (MIMD). In the SIMD approach, multiple processors perform similar operations on a large amount of data in a regular and possibly synchronized fashion. Examples of such (centralized) simulations are algorithmic mapping of *ANN* models on parallel SIMD machines [49], algorithmic mapping of feedforward neural

networks onto multiple bus systems [24], network learning on the connection machine [8], and systolic architectures for artificial neural networks [45]. These simulations are also known as data parallel approaches [71]. In mapping *ANN*'s onto MIMD machines, the *ANN* task is typically divided into several subtasks, each placed on a different computer. Each computer operates on its portion of the task and communicates with other computers through message passing schemes [71]. Examples of such mapping techniques include the simulation of the backpropagation algorithm on transputers [57]. Serbedzija [71] compares the performance of several parallel simulations of *ANN*'s on SIMD and MIMD computers. Next, we provide an overview of earlier studies on mapping of *ANN*'s on parallel computers.

1.3 Previous Work

Several studies on mapping *ANN*'s models onto parallel architectures have been reported in [24], [71], [51], [42], [50], [26], [49], [83], [28], [29], [45], [56], [37], [80] and [44]. Here, we provide a brief review of these works. El-Amawy and Kulasinghe [24] address the problem of mapping a feedforward *ANN* onto a multiple bus system with p processors and b buses. The proposed mapping scheme minimizes the total execution time of the learning algorithm. The multiple bus architecture is selected because of its inherent ability to provide concurrent broadcast operations. The paper also introduces a more efficient variant of the scheme when overlapping computations and communications are possible. The authors show that there is a unique arrangement of bus interfaces such that the number of interfaces is minimum while the optimal time is reached. The advantages of the proposed scheme over checkerboarding [42] are stated.

The study in [71] provides an overview of the techniques and strategies introduced for simulating artificial neural networks on parallel architectures. The study focuses on different parallelization schemes and discusses their merits. It explores parallel implementation of *ANN*'s on both general-purpose computers and neurohardware. The study points out the significance of mapping techniques on general purpose parallel computers and evaluates the performance of various simulation paradigms. It also addresses the significance and the need for dedicated computers for neural processing known as neurocomputers. Merits of digital and analog technologies as avenues for neurocompures are stated.

In [51], a technique for mapping feedforward *ANN*'s and Hopfield *ANN* models on hypercubes is introduced. The proposed scheme initially constructs a parallel architecture called mesh-of-appendixed-trees (*MAT*'s) for a given *ANN*. Then, *MAT* is embedded into the hypercube [51]. The asymptotic complexity of this scheme is given as $O(\log N)$, where N is the size of largest layer in the *ANN*. However, this logarithmic time is obtained at the expense of $3N^2$ processors for an $N \times N$ *MAT*. In that approach mapping of a feedforward *ANN* with a maximum of N neurons per layer on a hypercube requires a target architecture of a particular size, $4N^2$. The paper does not address the mapping of the *ANN* model on a hypercube of arbitrary size. Furthermore, this work does not consider the impact of the granularity of parallelism on the efficiency of the simulated algorithm.

In [42] a technique called *checkerboarding*, for mapping backpropagation algorithm on hypercubes and related topologies is introduced. This scheme utilizes the embedding of a $\sqrt{P} \times \sqrt{P}$ grid on a hypercube of size $P = 2^{2k}$. Elements of each weight matrix are

partitioned among processors of the grid. Neuron activation functions are assigned to diagonal nodes of the grid. The checkerboarding scheme requires less communication time than the pattern partitionings scheme reported in [40] and the vertical sectioning scheme utilized in [1] and [86]. In vertical sectioning an equal number of nodes from each layer of a uniform network are assigned to each processor. For *non-uniform* networks, the relative performance of the three schemes remains the same [42]. In a non-uniform *FFANN*, the number of neurons per layer differs from one layer to another. Non-uniform networks are utilized in many applications. The extension of checkerboarding scheme to non-uniform networks needs further study. The key issue is the choice of a proper square grid. The study in [42] provides a simplified algebraic analysis of the speedup for different network partitioning methods assuming high processor use. As the authors note, the trade-off between processor use and speedup needs to be further studied. The authors also claim that the checkerboarding scheme is asymptotically optimal. In other words, it can achieve time complexity of $O(\log L + \log I)$, where L is the number of patterns and I is the number of neurons per layer in a uniform network. However, to achieve this lower bound a large number of processors is needed, $P = LI^2$ [42]. Hence, to achieve asymptotic optimallity the number of processors should be a quadratic function of network size. Furthermore, the study does not consider whether varying the degree of granularity could improve the efficiency.

Fujimoto, et al., [26] propose massively parallel architectures, such as a toroidal lattice architecture (TLA) and a planar lattice architecture (PLA) for simulating *ANN*'s. The parallel architecture is designed in two steps. Initially, a multilayer perception network is

mapped to a set of virtual processors connected with a PLA or a TLA. Then, the virtual processors are mapped to a set of physical processors [26]. The mapping is performed based on a load balancing algorithm. It is stated that the PLA can realize a completely planar lattice structure which is efficient for WSI implementations [26]. The proposed load balancing imposes some limitations on the type of network which can be mapped by this scheme.

Lin, Prasanna, and Wojtek describe parallel implementations of *ANN*'s on fine grain mesh-connected SIMD arrays [49]. Their mapping scheme can be applied to those *ANN* models of arbitrary network topologies whose retrieval and training phases can be performed as matrix and vector manipulations [49]. The mapping is designed only for mesh-connected array processors. The authors introduce two versions of the mapping scheme. One version is designed for implementing *ANN* models with problem sizes smaller than the parallel machine size. In other words, the first version of the scheme can simulate an *ANN* with n neurons and e connections on an $N \times N$ mesh-connected SIMD machine if $n + e \leq N^2$. The other version uses a partitioning technique for implementing *ANN*'s of arbitrary size on a fixed size processor array. For mapping an *ANN* model with n neurons and e connections onto a $P \times P$ array, an $O(k)$ memory is required per processor, where $1 \leq k \leq \frac{N}{P}$. The study does not consider the possibility that a smaller sub-array of the fixed size SIMD architecture may lead to a more efficient simulation.

Wah and Chu [83] introduce a heuristic mapping of a multilayer *ANN* on a multicomputer system. They derive several results for reducing the complexity of the mapping process. Their mapping is based on the fact that neurons in a multilayer *ANN* can

be grouped in a number of clusters such that neurons in two connected clusters are pairwise connected [83]. They consider the computation time to be the dominant factor in the training phase of feedforward *ANN* models (This hardly reflects reality on any existing parallel machine). Based on this consideration, they decompose the physical processors into partitions in such a way that the error deviation of the heuristic approach from the optimal approach can be bounded [83]. They provide experimental results to confirm their claims. Their simplified model fails in cases where the communication time is not negligible (which is typically the case).

Ghosh and Hwang [29] investigate critical issues in mapping *ANN*'s on message-passing multiprocessors and develop a broad set of guidelines for efficient mapping of *ANN* models on parallel systems. They develop a structural model of an *ANN* by partitioning its network topology into groups of highly interconnected neurons. This model is used to find a proper set of guidelines for a heuristic mapping and to determine the functional behavior of neurons. They provide two principles for their heuristic approach, a partitioning principle and a mapping principle. They estimate the communication bandwidth required for balancing the communication and processing demands based on the structural model and the mapping policy [29]. They examine suitability of different classes of parallel architectures for simulation of *ANN* models. They indicate that architectures with direct links, such as hypercubes, multiple bus systems [64], and architectures with constant degree such as hypernets [38] or cube-connected cycles are more suitable than multistage networks. As we mentioned earlier, their work only provides a set of guidelines for developing a heuristic mapping scheme.

Kung and Hwang [44] develop a mapping scheme for implementation of *ANN* models on a linear systolic array of processors. Their scheme is based on the fact that the training phase of a feedforward ANN model can be expressed in terms of a sequence of matrix and vector operations. Wah and Chu [83] show that the scheme in [44] is optimal when the network is a feedforward *ANN*, and when the interconnection network is fast. The scheme in [44] however is only applicable to linear systolic arrays.

1.4 Research Objectives

Artificial neural networks with their huge computing power are primary candidates for solving large scale intractable scientific problems. The impressive processing capabilities of these models are due to their massively distributed structures, their ability to learn and generalize, and to their self-organizing and adaptive nature. These models offer a new processing paradigm which can be more powerful, robust, and user-friendly than conventional approaches [43]. Although *ANN*'s are abstract simulations of the real nervous system, we still have a long way to go before we can design a fully operational neurocomputer which resembles the human brain [35].

In the most general sense, our objective in this research is to develop a formal methodology for efficient mapping of contemporary *ANN* models on a popular class of parallel architectures. We consider parallel computers based on k-ary n-cube topologies since they encompass both mesh-connected and hypercube-based parallel systems. Many existing parallel architectures are based on these network topologies. These include MASPAR, nCUBE, Connection Machine, the Mosaic, Cray T3D, and the J-Machine [10]. We refer to parallel systems based on k-ary n-cube topologies as *KNC* architectures

henceforth. Our mappings will be designed to efficiently simulate an *ANN* model of arbitrary size on a *KNC*. The problem of mapping a specific *ANN* model onto a parallel architecture has been studied in the literature [24], [71], [51], [42], [50], [26], [49], [83], [28], [29], [65], [45], and [44]. Our approach has several significant features which distinguish it from earlier works. We elaborate on these features next.

The first appealing feature of our study is the wide range of *ANN* models we consider. Most studies on mapping *ANN*'s onto parallel architectures only considered one or two *ANN* models. In fact, most studies in this field were centered on the feedforward *ANN* with the backpropagation training algorithm [24], [51], [42], [28]. The study in [51] also covered the Hopefield model. We do not restrict our scope to a particular *ANN* model. Rather, we study a wide range of *ANN* models and develop a unified mapping approach for different classes of *ANN* models. The classification we utilize groups *ANN*'s based on similarities of their computational structure. Although specific implementations might vary from one model to another within a class, general mapping steps are similar for a given class of *ANN*'s. We present a systematic mapping scheme for each class and show how the mapping can be applied to specific *ANN*'s within that class. This feature of our study should have a significant impact on the study of *ANN*'s. With the availability of efficient implementations of wide range of *ANN*'s at their disposal, researchers in the field can effectively study different aspects of neural processing in order to develop powerful problem solving tools. Notice that no single *ANN* model is appropriate for every application. By providing efficient implementation of a range of *ANN*'s for users, the practicality of these models in engineering problems is significantly enhanced.

Another significant aspect of our study is that it covers an important class of *ANN*'s called *unit-allocating* neural networks. This class includes several important *ANN* models such as the *cascade correlation* [25] learning algorithm and the *adaptive resonance theory 1 (ART1)* [31] model. The common feature of these models is the dynamic nature of their architecture which grows during the learning phase. Hardware implementation of these models are difficult due to their dynamic structure. For instance, the cascade correlation training algorithm results in a network of cascaded neurons which cannot be implemented easily using VLSI technology [64]. Software simulation of such *ANN*'s seems to be a viable alternative for utilizing these powerful models. We develop a systemic and formal methodology for efficiently implementing these unit-allocating *ANN*'s on existing parallel systems. Based on our approach, no data migration or task reassignment is needed as number of hidden neurons grows during the training. To our knowledge, this has not been previously attempted perhaps due to the dynamic nature of the network architecture.

Learning speed is an important performance measure for *ANN*'s. Training algorithms are generally slow in nature requiring a large number of iterations to converge. One of the most important mapping objectives is to ensure that *ANN*'s are efficiently simulated on the given parallel architecture. The amount of parallelism achieved depends on the decomposition of the *ANN* model on the target architecture. To achieve an efficient simulation, it is essential to choose an appropriate granularity for partitioning the *ANN* model. We propose a unique approach which systematically picks an appropriate degree of parallelism which leads to a highly efficient realization of the *ANN* model on the target architecture. Our mapping scheme takes into account several factors for determining the

most suitable task granularity. Among these factors are the computational structure of the *ANN* model and the characteristics of the target parallel architecture including the computation time for atomic arithmetic operations and per word communication time between adjacent processors. If necessary, the scheme may only utilize a subset of the processors of a given *KNC* architecture (referred to as *subcube* henceforth). In such cases, the simulation on a subcube of the target architecture leads to the most efficient simulation.

The problem of determining the proper amount of parallelism for mapping an *ANN* model has not been addressed in earlier studies. Most studies impose some restrictions on the size of the *ANN* model or the target architecture. For instance, the mapping scheme introduced in [51] can efficiently simulate a feedforward *ANN* on a hypercube of size $4N^2$ only if the maximum number of nodes per layer in the *ANN* model is N. The scheme introduced in [42] maps a feedforward *ANN* on hypercube-based architecture of arbitrary size. However, to achieve asymptotic optimallity, the number of processors required is a quadratic function of the input size. Our scheme on the other hand does not impose any restrictions on the *ANN* model or on the target architecture.

1.5 Outline of the Dissertation

In Chapter 2, a formal methodology for optimal implementation of the backpropagation and similar algorithms on k-ary n-cube (*KNC*) topologies is presented. We first consider mapping of a feedforward artificial neural network (*FFANN*) models trained with the backpropagation algorithm on a *KNC* architecture. The methodology is developed by generalizing the optimal mapping of a bipartite graph. Initially, the *FFANN* is mapped onto a virtual *KNC*. The extent of parallelism is such that the simulation of the

learning pass on the virtual *KNC* is time optimal. Then, the virtual *KNC* is recursively folded until its dimension matches that of the physical architecture or a subcube thereof, depending on the physical size that provides the best execution time. A systematic folding process is developed to minimize execution time of each learning pass and to preserve the degree of redundancy. We prove that our mapping methodology is time-optimal and that it provides for maximum processor utilization regardless of the structure of the *FFANN*.

We show that the methodology developed for *FFANN*'s can be applied to several other classes of *ANN*'s. In particular, we consider several training algorithms for training *Radial Basis Function* (*RBF*) networks. We show that the training algorithms for these networks correspond to computational structures which are similar to those of the backpropagation algorithm.

In Chapter 3 we consider several unit allocating networks. A unit allocating network is one whose topology is modified during the training. We first consider the mapping of the Cascade Correlation learning algorithm. Cascade correlation [33] is an efficient supervised learning technique for neural networks. The learning algorithm incrementally adds and trains hidden units to a minimal topology until a desired error bound is reached. The significant attributes of such a "unit-allocating" network are fast learning (with polynomial time complexity) and compact representation of data [64]. The resulting architecture is a multi-layer network with cascaded single-unit hidden layers. VLSI implementation of this structure is difficult due to its irregular connections and unbounded fan-in [64].

The chapter first presents a methodology for efficient parallel implementation of the Cascade Correlation algorithm on k-ary n-cubes (*KNC*'s). We develop a computational model which captures the inherent parallelism of output-unit and hidden-unit training phases of the algorithm. Moreover, our model allows pipelining of several training patterns in order to further improve the efficiency of the implementation. The model we develop can easily be adapted to various parallel topologies. The mapping is done in two phases. The computational model is first mapped onto a virtual *KNC* of compatible size denoted by *VKNC*. Then, the *VKNC* is folded repeatedly (as necessary) such that a certain metric is optimized for a network with a certain number of hidden units. The folding is repeated until the resulting size is less than or equal to the size of the actual *KNC*. In the Cascade Correlation algorithm the number of hidden units is not known in advance. To efficiently map the training of such a dynamic network, we consider an upper bound on the number of hidden units (*Hmax*) . We consider two optimization criteria defined based on 1) the execution time of the algorithm for a network with *Hmax* hidden units and 2) the sum of execution times of the algorithm for all instances of the network with 0 through *Hmax* hidden units. We propose efficient analytical schemes for the mapping based on each criterion.

In the same chapter, we use the parameters for the benchmark application NETTalk to evaluate the performance of our mappings. We present experimental results which show that our approach leads to near-optimal results for networks with H hidden units where, $H < Hmax$. In addition, we show that the proposed scheme leads to very efficient simulation of the training algorithm even if the number of hidden units exceeds *Hmax*. We

also examine the effect of *Hmax* choice on the mapping. The minimization of each metric (assuming *Hmax* hidden units) has computational complexity $O\left(\log_k(L + Hmax)\right)$, for a network with L output units. Based on the proposed mapping, task assignments for networks with 0 through *Hmax* hidden units are known apriori. Hence, no data transfer or task rescheduling is needed as the number of hidden units grows.

Also in Chapter 3 we consider the mapping of a popular clustering network called Adaptive Resonance Theory (*ART*) [31]. We show that the mapping of this algorithm is very similar to that of the Cascade Correlation training algorithm. We provide simulation results for an efficient mapping of a benchmark example for this case as well.

Chapter 4 contains the conclusions drawn from this work and discusses some open problems for future work.

CHAPTER 2

MAPPING OF FEEDFORWARD AND RADIAL BASIS NETWORKS

In this chapter we introduce a methodology for optimal implementation of multi-layer *feedforward artificial neural networks* (*FFANN*'s) trained with the backpropagation training algorithm on *KNC*'s. The mapping is based on generalizing the mapping of a bipartite graph onto the *KNC* architecture. Initially, the *FFANN* is mapped onto a virtual *KNC* such that simulation of the learning pass on the virtual *KNC* is time optimal. The virtual *KNC* is then recursively folded until its dimension matches that of the physical architecture or a subcube thereof, depending on the physical size that provides the best execution time. A systematic folding process has been developed to minimize execution time of each learning pass. We prove that our mapping methodology is time-optimal and that it provides for maximum processor utilization regardless of the structure of the *FFANN*. The mapping scheme utilizes the given *KNC* architecture to achieve minimum execution time for both uniform and non-uniform *FFANN*'s. By considering the ratio of communication time to computation time per basic operation a proper subcube of the given *KNC* architecture is utilized to obtain the best granularity of parallelism in terms of minimization of total execution time.

We also address the efficient mapping of *Radial Basis Function* neural networks (*RBF*'s) on *KNC*'s. We consider both fully supervised and partially supervised training of *RBF*'s in our mapping. We show that the mapping of fully supervised training of *RBF*'s is very similar to that of a two-layer *FFANN* trained with the backpropagation algorithm.

Partially supervised training of *RBF*'s consists of two major phases. We introduce an efficient scheme for simulating the first phase. We show that the second phase can be realized as a two-layer *FFANN*.

2.1 Preliminaries

In this section we briefly review feedforward artificial neural networks, the backpropagation training algorithm, k-ary n-cubes, and some definitions from graph theory.

2.1.1 Feedforward Artificial Neural Networks

Feedforward *ANN*'s (*FFANN*) belong to a popular class of *ANN*'s used in many classification and pattern recognition applications. The backpropagation algorithm is a common learning algorithm used in training *FFANN*. The basic algorithm known as gradient-descent backpropagation [69] is very computation intensive and converges very slowly. Two common approaches to improve the learning speed of the algorithm are parallel implementations and the use of efficient variants of the algorithm. There are three types of parallelism associated with the backpropagation algorithm: algorithmic parallelism, spatial parallelism, and training parallelism [40]. Algorithmic parallelism exploits the parallelism intrinsic to the algorithm itself. Spatial parallelism on the other hand is related to the concurrency within a particular layer of a *FFANN* during the forward or backward phase of the algorithm [40]. Training parallelism (or pattern partitioning) is closely related to *off-line* training [40] in which weight increments are obtained for all training patterns before any weights are updated. In this case, the training set is divided into several subsets which are processed concurrently [40].

Several efficient variants of the algorithm are presented in [40]. In particular, a variant of the algorithm is the second-order least squares backpropagation [41] which is more complex than the basic algorithm but is more suitable for parallel implementation [41]. Other approaches are based on reducing the connectivity of the *FFANN* or on certain selection of the learning rate and the momentum term so as to reduce oscillation and speed up the convergence time.

FFANN's have been used in different applications. One common application of these networks is in continuous or discrete classification. Neurons in an *FFANN* are arranged into layers. In this work we consider a fully connected *FFANN* in which neurons in layer l, $0 < l < L$, are fully connected to neurons in layers l-1 and l+1 in an L-layer network. Each link connecting two neurons is assigned a weight called synaptic weight.

Generally, an *FFANN* consists of an input layer, one or more intermediate (hidden) layers, and an output layer. Figure 2.1 shows a fully connected L-layer *FFANN*. In this dissertation, the synaptic weight of the link between neuron j in layer l-1 and neuron i in layer l is denoted by w_{ij}^{l} for $1 \le i \le L$. The input vector to the network is an n-bit vector $X = \{x_1, x_2, ..., x_n\}$, and the output is an r-bit vector $Y = \{y_1, y_2, ..., y_r\}$. The output of neuron i in layer l, $1 \le l \le L$, is denoted by o_i^l and is computed by:

$$o_i^l = f\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l o_j^{l-1}\right) \forall\, i, 1 \le i \le N_l \qquad (2.1)$$

where o_j^{l-1} is the output of the j^{th} neuron in layer l-1, and $f(.)$ is the activation function. o_i^0 denotes bit i of the input pattern, x_i. Given a sample set $S = \{(X_1,Y_1), (X_2,Y_2), ..., (X_s,Y_s)\}$

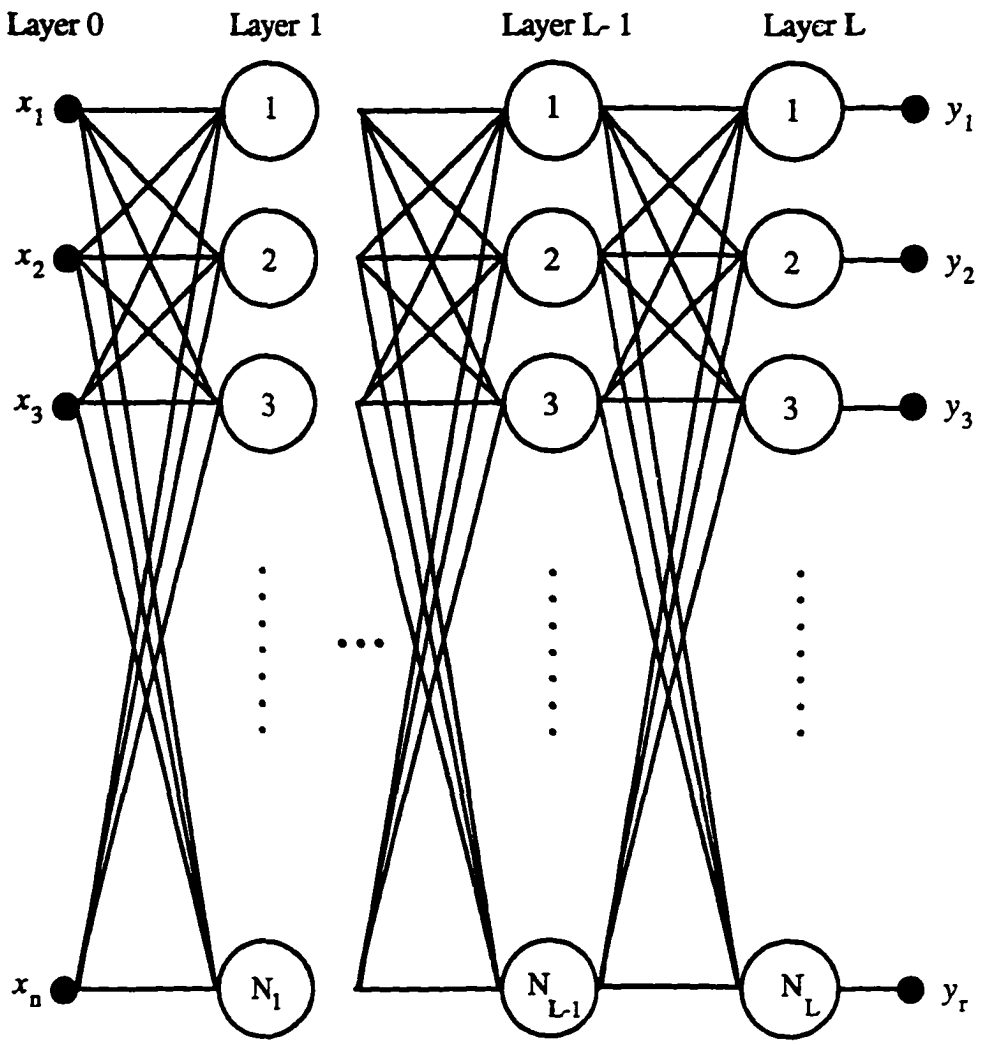

Figure 2.1: A generic feedforward *ANN*.

defined on $R^n \times R^r$, the training of an *FFANN* is defined as the search for a set of synaptic weights which can map the input patterns to their corresponding output patterns.

2.1.2 Error Backpropagation Training Algorithm

The *error backpropagation* training algorithm [69] (refereed to as backpropagation henceforth) is a supervised learning algorithm which is commonly used in training *FFANN*'s. It consists of two passes. In the forward pass each neuron j computes a weighted sum of its inputs, commonly referred to as NET_j. Then, the output of each neuron is computed by:

$$o_j = f(NET_j) \tag{2.2}$$

where f (.) is the activation function. At the end of the forward pass the computed and desired outputs of output layer neurons are used to calculate the error. The error function for the backpropagation training algorithm is expressed as follows:

$$E = \frac{1}{2} \sum_{j=1}^{N_L} [\, y_j^L - o_j^L \,]^2 \tag{2.3}$$

During the backward pass, the synaptic weights are adjusted so that the total error is minimized. The synaptic weights are updated using gradient-decent optimization [69]. The complete derivation of weight adjustments for a basic backpropagation algorithm is given in [69]. The weight updates can be obtained as follows:

$$\Delta w_{ij}^l(m) = \eta\ \delta_i^l\ .\ o_j^{l-1} + \alpha\, \Delta w_{ij}^l(m-1) \tag{2.4}$$

where $\Delta w_{ij}^l(m)$ is the weight adjustment for w_j^l during the m^{th} backward pass, η is the learning rate parameter, δ_i^l is the partial derivative of the associated error (or the *error term* for short) for neuron i in layer l, and α is the momentum term. The momentum term is added to speed up the learning without leading to oscillation [69]. For output neurons, δ_i^L is computed by

$$\delta_i^L = (\, y_i - o_i^L \,)\ f'\Big(\sum_{j=1}^{N_{L-1}} w_{ij}^L .\ o_j^{L-1}\Big) \tag{2.5}$$

and for hidden layer neurons it is computed as

$$\delta_i^l = f'\Big(\sum_{j=1}^{N_{l-1}} w_{ij}^l .\ o_j^{l-1}\Big)\ .\ \Big(\sum_{j=1}^{N_{l+1}} \delta_j^{l+1} .\ w_{ji}^{l+1}\Big) \tag{2.6}$$

The backpropagation algorithm is listed below:

Algorithm 2.1:

/* The backpropagation algorithm */

1. Initialize Weights.

2. If the error of output neurons are within acceptable range for every pattern, quit.

3. For every pattern $s \in \{(X_1,Y_1),...,(X_T,Y_T)\}$ do:

3.1 For every layer $l = 2, 3,...,$ L do

For every neuron $i = 1,...,N_l$ in layer l do

Compute the output of neuron i as follows:

$$o_i^l = f\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l o_j^{l-1}\right)$$

Endfor

Endfor

3.2 Compute Error Terms as follows

For the output layer neurons:

$$\delta_i^L(s) = f'\left(\sum_{j=1}^{N_{L-1}} w_{ij}^L \cdot o_j^{L-1}\right) \cdot \left(y_i(s) - o_i^L(s)\right)$$

For other layers $l = L\text{-}1, L\text{-}2, ..., 2$ use the backpropagated error as follows:

$$\delta_i^l(s) = f'\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l \cdot o_j^{l-1}(s)\right) \cdot \sum_{q=1}^{N_{l+1}} \delta_q^{l+1}(s) \cdot w_{qi}^{l+1}$$

4. Compute the Accumulated Weight Updates for layers $l = 1, 2, ..., L\text{-}1$:

$$\Delta w_{ij}^{l}(m) = \eta . \sum_{s=1}^{T} \delta_i^{l+1}(s) \, . \, o_j^{l}(s) + \alpha . \, \Delta w_{ij}^{l}(m-1)$$

5. Update Weights for $l = 1, 2, ..., L\text{-}1$ as follows:

$$w_{ij}^{l} = w_{ij}^{l} + \Delta w_{ij}^{l}(m), \text{ where } i=1,...,N_{l+1} \text{ and } j=1,...,N_l$$

6. Go back to step 2.

2.1.3 The *k*-ary *n*-cube Parallel Architecture

Several existing parallel computers such as Ametech 2020, n-cube1, n-cube2, Mosaic, iWarp, and the Cray T3D are based on *k*-ary *n*-cube (*KNC*) topologies [10]. Recent studies on the *KNC* are reported in [10],[27]. We briefly review general properties of *KNC*'s. The size (number of nodes) of an *n*-dimensional *KNC* is k^n, and each node can be uniquely identified by an *n*-digit *k*-ary label. The degree of each node is $2n$. The diameter of the network is $n \lfloor k/2 \rfloor$. Let $<e_{n-1},e_{n-2},...,e_0>$ be the *k*-ary address of an arbitrary node in the *KNC*. This node is connected to every node $<e'_{n-1},e'_{n-2},...,e'_0>$, where there exists only one i such that $e_i = (e'_i \pm 1) \bmod k$, while for all other digits $e_i = e'_i$. We make the following assumptions regarding the parallel architecture:

- A simple arithmetic operation such as addition, subtraction, or multiplication takes t_r units of time on a single processor of the *KNC*[1]. We refer to such operation as an *atomic* task.
- Each processor can communicate over one link at a time (single-port communication).

---

[1] This assumption can be easily changed to allow each operation to take a distinct time. It is intended here to simplify the expressions.

- *Per word* communication time between two adjacent processors on the *KNC* is t_c.
- Each communication unit is one *word*.
- Communication and computation can be overlapped.
- $t_r \leq t_c$ (This is the case in most if not all current machines).

Now, we introduce a binomial spanning tree (*BST*) of a *KNC* which is utilized in this work for optimal communications. A 0-dimensional k-ary binomial tree has one node. A 1-dimensional k-ary binomial tree is obtained by connecting k nodes in a linear array format and selecting the $\lfloor k/2 \rfloor^{th}$ node as the root. A 2-dimensional k-ary binomial tree is constructed out of k, 1-dimensional trees by connecting their roots as nodes of a 1-dimensional binomial tree. Figure 2.2 shows 1 and 2-dimensional 6-ary binomial trees. An n-dimensional k-ary binomial tree is constructed out of k, (n-1)-dimensional trees by connecting their roots as nodes of a 1-dimensional binomial tree. The height of the tree is clearly $n \lfloor k/2 \rfloor$. We assume that the root is at level 0.

We adopt the following labeling scheme for a *BST*. The scheme we introduce here is for an odd k. It can be easily modified for an even k. In a 1-dimensional k-ary *BST*, each node has a one digit label. The root gets address 0. Its left and right children are labeled k-1 and 1, respectively. The address of any other node on the left (right) subtree is obtained by decrementing (incrementing) the address of its parent. In an n-dimensional *BST*, the address of each node in any of the k, (n-1)-dimensional subtrees is obtained by appending the address of the root of the subtree (in 1-dimensional *BST*) to the left of its (n-1)-digit node address. This is illustrated in Figure 2.2.

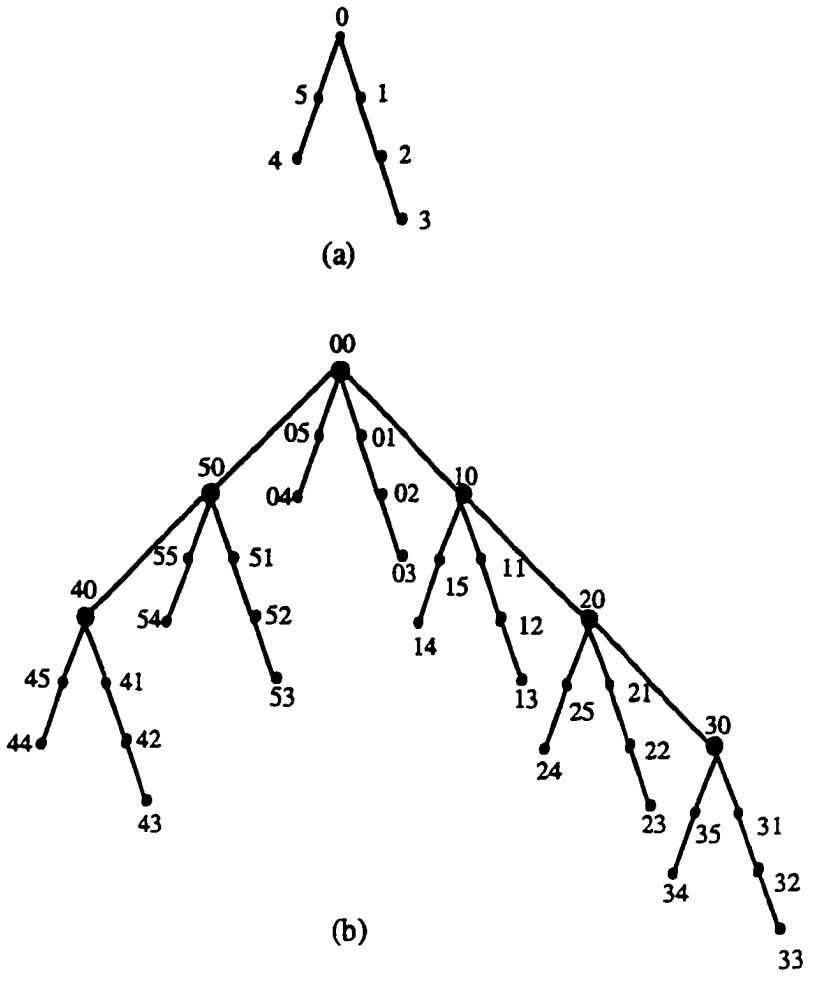

Figure 2.2: 6-ary binomial trees: a) 1-dimensional, b) 2-dimensional.

2.1.4 Some Preliminaries in Graph Theory

Let $G=(V_G,E_G)$ be a graph with p vertices and q edges. A non-empty graph is a graph with a non-empty vertex set. We denote the edge between any two adjacent vertices u and v by (u,v).

Definition 2.1: Given a non-empty graph $G=(V_G,E_G)$, the *line graph* $L(G)$ of G is defined as that graph whose vertices have a one-to-one correspondence with edges of G in such a way that any two vertices of $L(G)$ are adjacent if and only if their corresponding edges in

G are adjacent [19]. Note that two edges are considered adjacent if they have one vertex in common.

Definition 2.2: A mapping $\Phi: V(G_1) \rightarrow V(G_2)$ is called an *elementary contraction* [19] for two graphs G_1 and G_2 if G_1 contains two adjacent vertices u and v such that:

a) $\Phi(u) = \Phi(v)$, and $\{u_1,v_1\} \neq \{u,v\}$ implies $\Phi(u_1) \neq \Phi(v_1)$,

b) $\{u_1,v_1\} \cap \{u,v\} = \phi$ implies $(u_1,v_1) \in E(G_1)$ if and only if $(\Phi(u_1),\Phi(v_1)) \in E(G_2)$,

c) If $w \in V(G_1)$ and $w \neq u,v$ then $(u,w) \in E(G_1)$ or $(v,w) \in E(G_1)$ if and only if $(\Phi(u),\Phi(w)) \in E(G_2)$.

We denote the graph obtained from identification of adjacent vertices u and v in graph G by $EC_{uv}(G)$.

Definition 2.3: A graph G_1 is isomorphic to graph G_2 if there exists a one-to-one mapping Φ, called isomorphism, from $V(G_1)$ to $V(G_2)$ such that Φ preserves adjacency [19].

Definition 2.4: A contraction $C: V(G_1) \rightarrow V(G_2)$ is a mapping that is either an isomorphism or a composition of finitely many elementary contractions [19].

2.2 Optimal Mapping of *FFANN*'s

In this section we introduce a scheme for optimal mapping of *FFANN*'s on k-ary n-cube multiprocessors. The objective is to map the computations of the *FFANN* to physical processors of the *KNC* such that the execution times required by both the training and retrieval phases of the *FFANN* are minimized. Our mapping scheme is based on network partitioning. However, we still utilize characteristics of the training algorithm to simplify our mapping procedure.

The nature of the backpropagation training algorithm does not allow the computations corresponding to a given layer of an *FFANN* to proceed unless the computations corresponding to the previous layer have been completed. During the forward pass of the algorithm, computations of layer l precede those of layer l+1, where $1 \le l \le L$-1. On the other hand, computations of layer l follow those of layer l + 1 during the backward pass. This implies that for a particular pattern, the overlap of execution of tasks in different layers of an *FFANN* is not possible during the forward or backward passes of the backpropagation training algorithm. Thus, the problem of mapping an *FFANN* reduces to that of mapping a set of bipartite graphs (or *BPG*'s for short) representing (overlapping) pairs of adjacent layers of the *FFANN* and their synaptic links. Since mapping of a certain *BPG* would be very similar to that of another, we can simplify the problem further by considering a virtual *BPG* that is large enough to accommodate any of the *BPG*'s corresponding to adjacent layers.

The key issue here is to determine the size of each partite set of the virtual *BPG*. Let l_{omax} and l_{emax} be the indexes of the largest layers among odd-indexed and even-indexed layers of the *FFANN*, respectively. Notice that these layers need not be adjacent in the *FFANN*. The sizes of these two layers are denoted by N_{omax} and N_{emax}, respectively. It is clear that such a graph can accommodate any two adjacent layers in the given *FFANN* in the sense that it is a supergraph of the graph representing these layers. We will refer to such a *BPG* as the *largest virtual layer* graph (or *LVL* for short). Figure 2.3 shows the *LVL* graph for a generic *FFANN*. Conceptually, an L-layer *FFANN* can be considered as L overlapping *BPG*'s. Each such *BPG* consists of two adjacent layers of neurons and their connecting

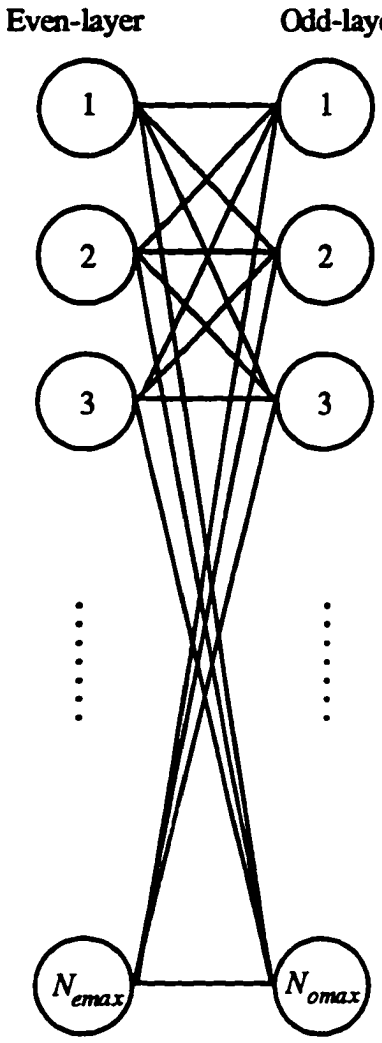

Figure 2.3: The largest virtual layer.

synaptic links. Such graphs can be also used to represent the concurrent communications and computations involved during the forward pass or the backward pass of the backpropagation algorithm. We denote the *BPG* representing layers l and l+1 and their corresponding links by BPG_l. Figure 2.4 illustrates *BPG*'s of a generic L-layer *FFANN*.

After obtaining the *LVL*, our mapping methodology consists of three steps. Initially, an optimal processor assignment is obtained for the *LVL* on a virtual *KNC* which we refer to as *VKNC* henceforth. The mapping of the *FFANN* onto the *VKNC* is then obtained by generalizing that of its *LVL*. In particular, the mapping scheme obtained for the *LVL* is

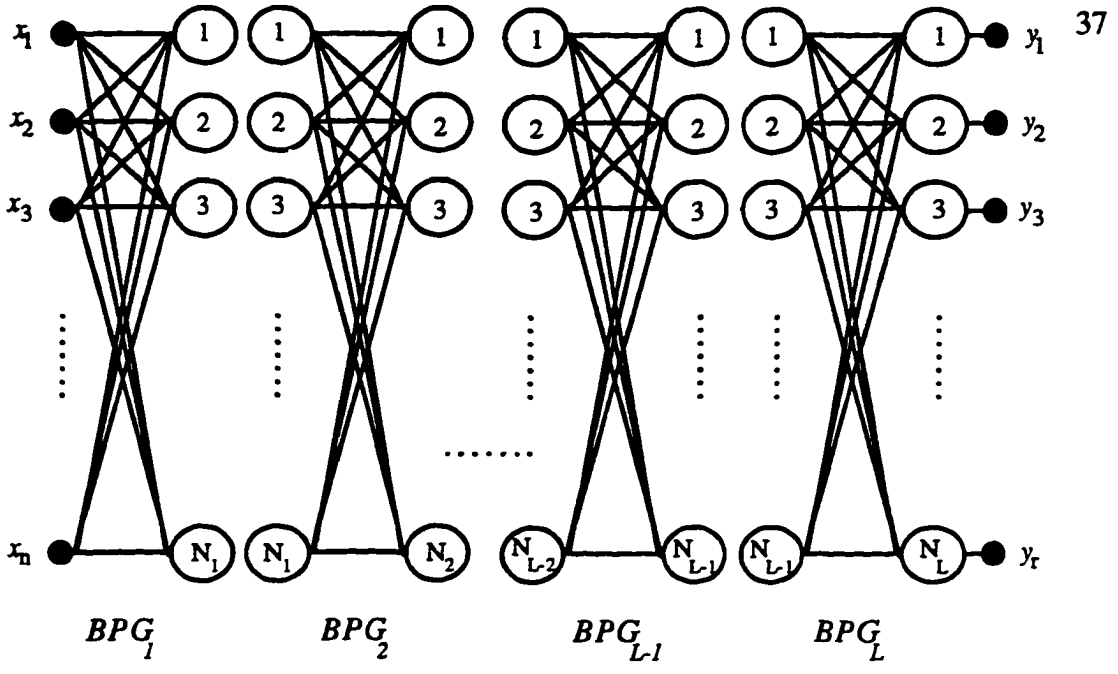

Figure 2.4: *BPG* graphs of an *FFANN*.

applied to every *BPG* of the *FFANN*. Finally, the *VKNC* is contracted (or folded) to fit the actual *KNC* or a subnetwork thereof based on the time optimality requirement. The contraction of *VKNC* is performed such that the learning is time optimal. We will explain these steps in detail in the following subsections.

2.2.1 Mapping the *LVL* onto a *VKNC* Architecture

Here we describe a scheme for mapping the *LVL* associated with a given *FFANN* onto a *VKNC* architecture. Initially, we obtain the line graph of the *LVL*, denoted by $LG = (V_{LG}, E_{LG})$ (see Definition 2.1). Notice that every node of the *LVL* represents a pair of adjacent neurons; one from the even layer and one from the odd layer. We assign vertices of the *LG* to a *VKNC* with radix k and dimension $n_v = n_o + n_e$, where $n_o = \lceil \log_k N_{omax} \rceil$ and $n_e = \lceil \log_k N_{emax} \rceil$. Each of the k^{n_v} nodes of the *VKNC* can be uniquely identified by an n_v-digit address. Obviously, $|V_{LG}| \leq k^{n_v}$, where $|V_{LG}|$ is the size of the *LG*. Thus, the

VKNC is large enough to fit the *LG* in the sense that every vertex of the *LG* is assigned to a unique node of the *VKNC*. The assignment is done as explained below.

To identify each node in V_{LG}, we use the following labeling scheme. The label of a node in V_{LG} representing the synaptic link between neuron $<a^{n_o-1}, a^{n_o-2},..., a^0>$ in the odd-layer and neuron $<b^{n_e-1}, b^{n_e-2},..., b^0>$ in the even-layer of the *LVL* is obtained by concatenating the labels of the two neurons as follows:

$$< a^{n_o-1}, a^{n_o-2},..., a^0, b^{n_e-1}, b^{n_e-2},..., b^0 >$$

For convenience, we will separately denote the three components of the address of node v_i^j in V_{RLG} by $<A_iB_i>$, where

$$A_i = a_i^{n_o-1}, a_i^{n_o-2},..., a_i^0, \text{ and } B_i = b_i^{n_e-1}, b_i^{n_e-2},..., b_i^0$$

We refer to A_i and B_i as the *odd-range* and the *even-range* of the node address in the *VKNC*, respectively.

At this point, the n_v-digit label of each node in the *LG* graph directly specifies the processor in the *VKNC* to which that node will be assigned. Based on this assignment scheme a copy of neuron *J* from the odd-layer of the *LVL* is assigned to N_{emax} nodes of the *VKNC* with addresses $<A_iB_i>$, where $A_i = J$, and $0 \le B_i \le k^{n_e}-1$. Similarly, a copy of neuron *I* from the even-layer of the *LVL* is assigned to N_{omax} nodes of the *VKNC* with addresses $<A_iB_i>$ where $0 \le A_i \le k^{n_o}-1$ and $B_i = I$. Figure 2.5 shows a sample *LVL* and the assignment of its neurons to a 3-dimensional binary *VKNC*. The assignment of node $<IJ>$ of the *LG* to a node of the *VKNC* implies storing, at that node, (or in some cases initializing, as we explain later) the parameters of a 5-tuple (w_{IJ}, o_I, o_J, δ_I, δ_J) in the corresponding virtual processor. The parameters of the 5-tuple are define as follows: w_{IJ} is the synaptic weight

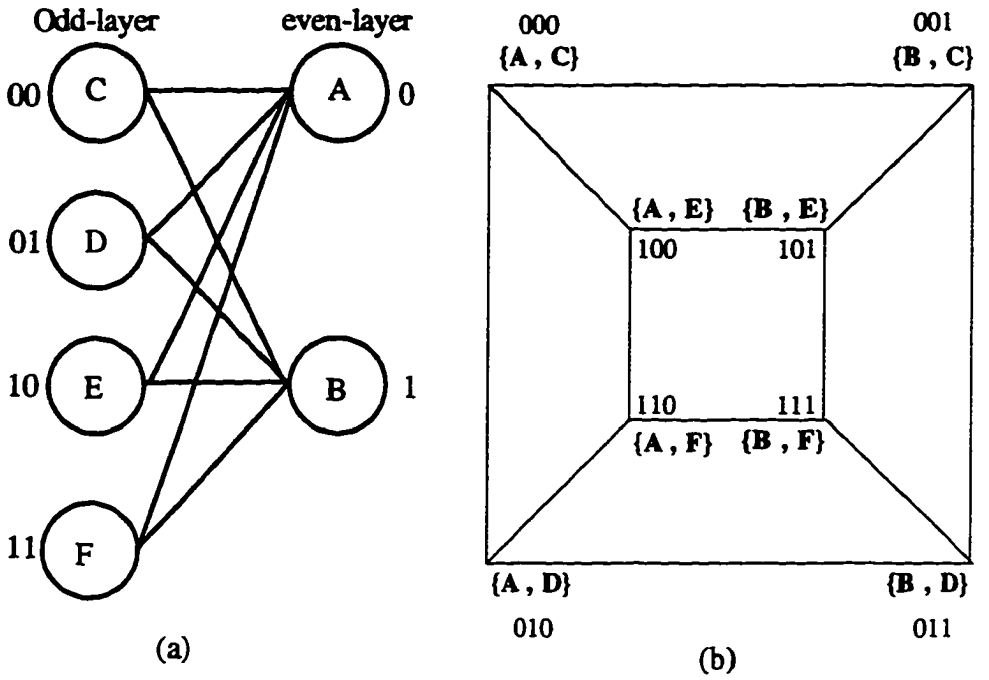

Figure 2.5: Processor assignment: a) a sample *LVL*, and b) its implementation on a 3-D hypercube.

between the two neurons I and J, o_I (o_J) is the output of neuron I (J), and δ_I (δ_J) is the error term mapped to neuron I (J). For simplicity we refer to this 5-tuple as $DATA_{IJ}$.

2.2.2 Applying the Assignment Procedure to the *FFANN*

The simulation of an *FFANN* onto the *VKNC* can be obtained through a straightforward generalization of the mapping of its corresponding *LVL*. The mapping procedure developed for the *LVL* is applied independently to each BPG_l ($1 \le l \le L$) of the *FFANN*. Basically, the assignment of neurons in the odd-layer (even-layer) of the *LVL* is applied to each odd-indexed (even-indexed) layer of the *FFANN*.

The assignment procedure should provide proper communications between adjacent *BPG*'s. Outputs of neurons in layer l, which are computed by the BPG_l, are used by BPG_{l+1} to compute outputs of neurons in layer l+1. Similarly, error terms associated with neurons in layer l, which are computed by the BPG_{l+1}, are used by BPG_l to compute the error terms

of neurons in layer l-1. The following assignment procedure is designed to provide for such communications.

Without loss of generality, assume that l is an odd number. Neurons in the odd-layer of BPG_l (layer l) are mapped to neurons in the odd-layer of the *LVL* such that neuron i from the odd-layer of BPG_l is mapped to neuron i from the odd layer of the *LVL*, for $1 \leq i \leq N_l$. Also, neurons in the even-layer of BPG_l (layer l-1) are mapped to neurons in the even-layer of the *LVL* in a similar manner. Note that by its definition the *LVL* can fit any BPG_l. The processor assignment for each neuron in BPG_l is the same as that of the neuron in the *LVL* to which it maps. Recall that the assignment for each neuron in the *LVL* was obtained by mapping the corresponding *LG* to the *VKNC*.

2.2.3 Optimal Simulation of Each Learning Pass on the *VKNC*

Both the forward and backward passes of the gradient-descent backpropagation algorithm include the computation of *sum-of-products* terms. These steps compute the output or the error term of each neuron during the forward and backward passes of the algorithm, respectively. Efficient implementation of these steps results in efficient overall execution of the algorithm. This is due to the fact that the communications and computations associated with these steps require significantly more time than other steps involved. Note that other steps can be computed locally in each processor without any communication overhead.

According to our processor assignment, each virtual processor is assigned a pair of adjacent neurons. During the sum-of-products computations, each processor computes one product term, the smallest possible task. Then, the sum of such products could be

computed using a fan-in algorithm involving all processors of the *KNC*. However, this might not be the time-optimal solution. We must find the granularity or the extent of parallelism which results in the optimal solution, given the specific times needed to perform a computation and to communicate.

We need to find the dimension of the *KNC* architecture which can compute the sum of products of k^n pairs optimally. We denote the dimension of such *KNC* by n^{opt}. Clearly, $0 \leq n^{opt} \leq n$. We shall utilize the *BST* rooted at an arbitrary node of the *KNC* to perform these computations in minimum time. We first obtain the optimal time for adding k^n numbers on an n-dimensional *KNC*. We adhere to the assumptions stated in Subsection 2.1.3 throughout our derivations.

Theorem 2.1: The minimum time to compute the sum of k^n numbers stored at the nodes of an n-dimensional *KNC* is $n\,(\lceil k/2 \rceil t_c + 2\,t_r)$ where t_c is the communication time between two adjacent processors, and t_r is the computation time on a single processor for a simple mathematical operation.

Proof: We use mathematical induction. Let $T(n)$ denote the time required for adding k^n numbers on an n-dimensional *KNC*.

Induction basis (T(1)):

Consider the case where k is even. We shall refer to the node at which the final sum resides at the *root*. We need to show that the sum of k numbers residing on nodes of a 1-dimensional *KNC* can be computed optimally in $T(1) = (k/2\,t_c + 2\,t_r)$ time units. The diameter of the 1-dimensional *KNC* is $\lfloor k/2 \rfloor$ [10]. Hence, at least $k/2\,t_c$ time units are required to get every value to the root. The minimum time required to add k numbers

(assuming that a processor can add two numbers in t_r units of time) is $\lceil \log_2 k \rceil t_r$ time units. This is accomplished by performing the addition in a binary tree fashion. To obtain the optimal addition scheme on the 1-dimensional *KNC* we have to find the maximum number of computational (addition) steps which can be overlapped with communication steps. Each overlapped step takes t_c units of time, since $t_c \geq t_r$. We show that under the given conditions, two computation steps cannot be overlapped with communication. One is clearly the final addition which is performed to obtain the final sum. Next, we show that the first addition step (in any algorithm) cannot be overlapped either.

Originally, each processor has one operand assigned to it. Hence, the first step of any algorithm involves the transfer of values from some processors to others. The first step takes t_c time units. Clearly, there exist processors at distances $k/2$, $k/2$ -1,..., and 1 from the root. We denote the processor at distance i from the root by P[i]. If this processor does not participate in the first transfer, it has to send its operand to the root in at least $k/2$ t_c time units. Thus, the problem would require at least ($k/2$ + 1) t_c + t_r time units which is greater than or equal to the stated bound.

On the other hand, if processor P[$k/2$] transfers its value during the first step, it can only send it to its adjacent processor denoted by P[$k/2$ -1], which is at distance $k/2$ -1 from the root. Now, we show that to perform the addition in minimum time, the second step of any algorithm should perform only addition. Assume that at least one processor performs a transfer during the second step. We consider two cases. First, assume that processor P[$k/2$ -1] is among the processors that communicate during the second step. Therefore, the second step takes t_c time units. Note that this processor has two operands after the initial

communication step, one received from processor P[$k/2$] and the one assigned to it originally. So, if processor P[$k/2$ - 1] sends one of the two values during the second step, it still needs $k/2 - 1$ communication steps to send the second operand to the root. Hence, in this case the problem would require at least $(k/2 + 1)\, t_c + t_r$ time units which is greater than or equal to the stated minimum.

Second, assume that processor P[$k/2$ -1] is performing addition, but some other processor is performing a communication during Step 2. Processor P[$k/2$ - 1] needs either at least $k/2 - 1$ communication steps to send the sum it has calculated during the second step to the root or at least $k/2$ steps to send its two operands. The latter case would take at least $(k/2 + 1)\, t_c + t_r$ units of time which is again greater than or equal to the stated bound. Thus, in Step 2 only addition should be performed. Hence, under the given conditions, the lower bound on computing the sum of k values on a 1-dimensional *KNC* is $T(1) = (k/2\, t_c + 2\, t_r)$ time units.

When k is odd, there are two nodes at distance $\lfloor k/2 \rfloor$ from the root. Since each processor can read from one port at a time, at least $\lceil k/2 \rceil$ communication steps are required. Similar to the case with an even k, there are two addition steps which cannot be overlapped with communications. Hence, the optimal addition scheme takes $T(1) = \lceil k/2 \rceil\, t_c + 2\, t_r$ time units when k is odd. Since $\lceil k/2 \rceil = k/2$ when k is even, the theorem is true for the base case.

Induction Step: (T(n) implies T(n+1)):

Assume $T(n) = n\ (\lceil k/2 \rceil\, t_c + 2\, t_r)$ is the minimum time to compute the sum of k^n values on an n-dimensional *KNC*. We show that $T(n+1) = (n+1)\ (\lceil k/2 \rceil\, t_c + 2t_r)$ is the

optimal time to add k^{n+1} values on an $(n+1)$ dimensional *KNC*. By definition, an $n+1$-dimensional *KNC* is obtained by connecting k n-dimensional *KNC*'s as nodes of a 1-dimensional *KNC*. By the induction hypothesis, the local sum at some node (call it *root*) of an n-dimensional *KNC* is computed optimally in $T(n) = n\ (\lceil k/2 \rceil t_c + 2\ t_r)$ time units. It remains to add the values stored at the roots of the n-dimensional *KNC*'s. These k roots form a 1-dimensional *KNC*. By the induction basis, the minimum time to add k values on a 1-dimensional *KNC* is $T(1) = \lceil k/2 \rceil t_c + 2\ t_r$ time units. Thus, the optimal time to add k^{n+1} values on an $(n+1)$-dimensional *KNC* is

$$\begin{aligned} T(n+1) &= T(n) + T(1) \\ &= n(\lceil k/2 \rceil t_c + 2\ t_r) + (\lceil k/2 \rceil t_c + 2t_r) \\ &= (n+1)(\lceil k/2 \rceil t_c + 2t_r) \end{aligned}$$

Thus the proof is complete. □

Next, we present a *fan-in* algorithm which adds k^n numbers on an n-dimensional *KNC* optimally when k is an even number. The algorithm can be easily modified for an odd k. We define *lsum*[i] to represent the partial sum stored in node i of the *KNC*. Initially, *lsum*[i] is equal to the value assigned to node i. The *fan-in* addition algorithm is listed below

Algorithm 2.2:

/* Fan-in Addition */

Begin:

For ($m = 1$ to n) Do

/ compute the sum in every m-dimensional BST and store it in its root */*

For ($p = 0$ to $k^{n-m} - 1$) Parallel Do

1. */* Add values in any two adjacent nodes and send the sum toward the root */*

For ($j = 0$ to $\lceil k/4 \rceil - 1$) Parallel Do

For (the left and right branches) Parallel Do

1.1. If (*left branch*) Do

send $lsum[pk^m + (k/2 + 2j)k^{m-1}]$

to $[pk^m + ((k/2 + 2j+1) \bmod k/2)\, k^{m-1})]$;

add $lsum[pk^m + (k/2 + 2j)k^{m-1}]$

to $lsum\,[pk^m + ((k/2 + 2j+1) \bmod k/2)\, k^{m-1})]$;

send $lsum[pk^m + (k/2 + 2j + 1)k^{m-1}]$

to $[pk^m + ((k/2 + 2j+2) \bmod k/2)\, k^{m-1})]$;

1.2. Else */* right branch */*

send $lsum[pk^m + (k/2 - 2j - 1)k^{m-1}]$

to $[pk^m + (k/2 - 2j - 2)\, k^{m-1}]$;

add $lsum[pk^m + (k/2 - 2j - 1)k^{m-1}]$

to $lsum\,[pk^m + (k/2 - 2j - 2)k^{m-1}]$;

send $lsum[pk^m + (k/2 - 2j - 2)k^{m-1}]$

to $[pk^m + (k/2 - 2j - 3)\, k^{m-1}]$;

Endfor

Endfor

2. */* Transfer partial sums toward the root of each m-dimensional BST, and if a value has arrived at the root add it to the root's partial sum */*

For ($i = 2$ to $k/2$) Do

For ($j = 0$ to $\lfloor \frac{k/2 - i}{2} \rfloor$) Parallel Do

For (*the root and the left & right branches*) Parallel Do

2.1. If (*the left branch*) Do

send *lsum*[$pk^m + (k/2 + 2j + 1)k^{m-1}$] from

node [$pk^m + (k/2 + 2j + i) k^{m-1}$]

to node [$pk^m + ((k/2 + 2j + i + 1) \bmod k/2) k^{m-1})$];

2.2. Else If (*the right branch*) Do

send *lsum*[$pk^m + (k/2 - 2j - 2)k^{m-1}$] from

node [$pk^m + (k/2 - 2j - i - 1)k^{m-1}$]

to node [$pk^m + (k/2 - 2j - i - 2) k^{m-1}$];

/* *The computations in the root* */

2.3. Else If (*the root*) Do

If (*k/2 and i are both even or both odd*) Do

add the *lsum* received from [$pk^m + (k - 1)k^{m-1}$] to *lsum* [pk^m];

Else Do

add the *lsum* received from [$pk^m + k^{m-1}$] to *lsum* [pk^m];

Endfor

Endfor

Endfor

End;

Lemma 3.1: The above fan-in algorithm is time-optimal.

Proof: We first obtain the time required by each step in the algorithm. Step 1 consists of Steps 1.1 and 1.2 which are performed concurrently. These steps involve 2 communication steps between adjacent nodes and an addition step. Hence, they take $2t_c + t_r$ time units. Steps 2.1 through 2.3 are executed in parallel as well. Steps 2.1 and 2.2 consist of a communication between adjacent processors while Step 2.3 involves a single addition. Hence, these steps can be done concurrently in t_c time units. The first $k/2$ -2 iterations of Step 2 involve concurrent execution of Steps 2.1 through 2.3. During the final iteration of Step 2, only Step 2.3 is executed. Hence, Step 2 takes $(k/2 - 2)t_c + t_r$ time units. The algorithm consists of n iterations of Steps 1 and 2. Therefore, the algorithm takes $T(n) = n\,(\,k/2\;\; t_c + 2\,t_r\,)$ time units, which is time-optimal. □

Next, we utilize Theorem 2.1 and Lemma 2.1 to develop an optimal method for computing the sum of products of k^n pairs on an n-dimensional *KNC*, i.e., each processor holds two values (or several pairs) which must be multiplied and the result added to corresponding results at all participating processors. Depending on the ratio of communication time to computation time, the optimal solution might utilize only a subcube of the physical architecture. Therefore, we will first obtain the minimum time required by the computation on an i-dimensional *KNC* assuming that all processors participate in the computation ($i \leq n$). Theorem 2.2 states this lower bound. Once we compute the lower bound, we will find the dimension of the subcube of the n-dimensional which provides the best execution time.

Theorem 2.2: The minimum time to compute the sum of products of k^n pairs on an i-dimensional *KNC* ($i \le n$) is

$$T(i) = i\left(\lceil k/2 \rceil t_c + 2\,t_r\right) + (2\,k^{n-i} - 1)\,t_r \qquad (2.7)$$

Proof: Let Q_j pairs be assigned to processor j, where $1 \le j \le k^i$ and $Q_j \ge 1$. It takes t_r time units to compute the first product. According to Theorem 2.1, the minimum time required to perform the fan-in addition for the first set of products is $T(i) = i\left(\lceil k/2 \rceil t_c + 2\,t_r\right)$ time units. Hence, the first set of products can be computed and added to the root in $T(i) = i\left(\lceil k/2 \rceil t_c + 2\,t_r\right) + t_r$ time units. The best possible scenario is one in which multiplication of the remaining pairs and the addition can be pipelined such that a new computation can begin every t_r time units. This results in the following lower bound for computation of the sum of products:

$$T(i) = \left(\max(Q_j)\right)t_r + i\left(\lceil k/2 \rceil t_c + 2\,t_r\right) \qquad (2.8)$$

However, we show that this lower bound cannot be achieved. To obtain a tighter lower bound, we use the notion of *reservation table* [77]. Each physical processor is considered a stage of the pipeline (used to multiply and add the pairs) and is assigned a row in the reservation table. Each column of the reservation table represents either a communication time unit or a computation time unit. The average delay between any two successive initiations of new computations in the pipeline cannot exceed the maximum number of marks in any row of the reservation table [77].

At least $(i \log_2 k)\,t_r$ time units are needed to add the first set of products. Hence, there must exist processors which perform at least one addition and one multiplication in the

same step(s), regardless of how multiplication or addition steps take place. Since this is not allowed, according to [77] the average delay between initiations must be at least $2t_r$. This indicates that it takes $(\max Q_j - 1)\, 2t_r$ time units to compute and fan-in the remaining products. Thus, we have

$$T(i) \geq (2\max(Q_j) - 1)t_r + i(\lceil k/2 \rceil t_c + 2t_r) \tag{2.9}$$

We need to find the minimum for the RHS of the above inequality by determining the best distribution of products among processors. Since $\sum Q_j = k^n$ and $Q_j > 0$, clearly min (max (Q_j)) is obtained when pairs are distributed uniformly among processors , i.e., min (max (Q_j)) = k^{n-i}. Hence, the optimal time is

$$T(i) = (2k^{n-i} - 1)t_r + i(\lceil k/2 \rceil t_c + 2t_r) \tag{2.10}$$

Thus the proof is complete. □

At this point, we can find the dimension ($i \leq n$) of the subcube of the physical *KNC* which provides the best execution time. In other words, we find n^{opt} such that $T(n^{opt}) = \min_{0 \leq i \leq n} T(i)$.

Theorem 2.3: Let $T(i)$ denote the minimum time to compute the sum of products of k^n pairs on an i-dimensional *KNC*. Then, n^{opt} is either one of the two end points of the interval $[0,n]$, or is given by either the ceiling or floor function of

$$n^{opt} = n - \log_k \left(\frac{\lceil k/2 \rceil t_c}{2t_r \ln k} + \frac{1}{\ln k}\right) \tag{2.11}$$

Proof: For now, assume i is a real positive number. Then, $T(i)$ is a continuous function on the set of real numbers (given by (2.10)). The first and derivative of $T(i)$ is given as follows:

$$\frac{\partial T(i)}{\partial i} = -2 t_r \ln k \; k^{n-i} + \lceil k/2 \rceil t_c + 2 t_r \tag{2.12}$$

In addition, the second derivative of $T(i)$ is given by

$$\frac{\partial^2 T(i)}{\partial i^2} = 2 t_r (\ln k)^2 \; k^{n-i} \tag{2.13}$$

According to [9], if c is an extreme point of this function, then one of the following must hold: i) $T'(c)$ fails to exist, and ii) $T'(c) = 0$. Since the first derivative exists for every i, the extreme is obtained by setting $T'(i) = 0$. The second derivative is always positive indicating that $T(i)$ is a convex function. Hence, $T(i)$ has one minimum which is given by:

$$i_{\min} = n - \log_k \left(\frac{\lceil k/2 \rceil t_c}{2 t_r \ln k} + \frac{1}{\ln k} \right) \tag{2.14}$$

We use the above result to find the minimum $T(i)$ in the closed interval $[0,n]$. Clearly, if i_{min}, obtained from (2.14), is in the closed interval $[0,n]$, then it is also the minimum for the function over the closed interval, i. e., $i_{min} = n^{opt}$. However, if $i_{min} < 0$ $(i_{min} > n)$ then $n^{opt}=0$ $(n^{opt}=n)$. This is due to the fact that $T(i)$ is a convex function.

For our purposes, n^{opt} must be an integer value in $[0,n]$. Hence, either the ceiling or the floor of the RHS of (3.9) results in minimum $T(i)$ for integer values of i in the given interval. □

We use Theorem 2.3 to find the optimal degree of parallelism for each layer. We divide the neurons of each layer into n_l^{opt} clusters, where n_l^{opt} is the optimal dimension of

a subcube of the *VKNC* executing a *sum-of-products* computation involving neurons in layer *l*. The size of each cluster for an even (or odd) layer is denoted by E_l^0 (or O_l^0), where

$$E_l^0 = k^{n_l - n_l^{opt}} \text{ for an even } l$$
$$O_l^0 = k^{n_l - n_l^{opt}} \text{ for an odd } l$$

The significance of E_l^0 (and O_l^0) will be clarified shortly. For simplicity, henceforth we refer to n_l^{opt} as n_l.

Once the overall sum is calculated for layer *l* during the forward pass (backward pass), it should then be broadcasted to every node of the subcube associated with neurons in layer *l*+1 (*l*-1) using an optimal one-to-all broadcast algorithm. According to [10], this task (during the forward pass) can be performed optimally in $n_{l+1}\ k/2\ t_c$ time units when *k* is even. For the case of odd *k* it is shown that the one-to-all broadcast can be performed in $n_{l+1}\lceil k/2 \rceil$ time units [10]. Notice that the size of the subcube used for such broadcasts is properly chosen for each layer to minimize the communication overhead. This maximizes the performance even for non-uniform *FFANN*'s.

2.2.4 Optimal Folding of the *VKNC*

The *VKNC* will generally be larger in size than the physical *KNC*. In this subsection, we introduce a procedure for partitioning nodes of the *VKNC* and assigning them to those of the physical *KNC*. The mapping procedure ensures optimal simulation of the *FFANN* network.

The assignment procedure is based on the topological structure of the *KNC*. An *n*-dimensional *KNC* structure contains *k* edge-disjoint (*n*-1)-dimensional *KNC*'s interconnected as vertices of a 1-dimensional *KNC*. There are *n* distinct ways to partition nodes of an *n*-

dimensional *KNC* into k, (n-1)-dimensional *KNC*'s obtained by partitioning along any of the n dimensions. By selecting any of the n digits and fixing its value to say r, an (n-1)-dimensional *KNC* is uniquely specified. Since r is such that $0 \leq r \leq k$-1, k distinct (n-1)-dimensional *KNC*'s are identified uniquely based on that digit.

In our scheme, the *VKNC* undergoes several contractions (see Definition 2.4) until its size becomes equal to that of the actual *KNC*. At each iteration of the contraction process, the dimensionality of the *VKNC* is reduced by 1, by identifying all nodes whose k-ary addresses differ only in a particular digit. We refer to this digit as the *folding digit*.

A set is associated with each node of the contracted *VKNC*, called *neuron set*, to represent all neurons assigned to the node. Originally, the neuron set of node *IJ* contains *DATA* (defined in Subsection 2.2.1) associated with any neuron in its cluster. After each contraction, the neuron set of each resulting node (obtained as a result of contracting k nodes of the original architecture) includes (the union of) neuron sets of the k identified nodes. Notice that during the t^{th} folding step, each k-node set of the *VKNC*'s vertices whose addresses differ in only one digit are contracted to one node called the *identified node*.

The key issue here is how to select the folding digit during each iteration of the contraction process to minimize the overall execution time. Obviously, $n_f = n_v - n_a$ iterations are required until the folded *VKNC* is equal in size to the actual *KNC*, where n_v and n_a are the dimensions of the *VKNC* and the physical *KNC*, respectively. Further contractions (see Definition 2.1.4) might be necessary to ensure the time optimality of the overall simulation. As shown earlier, the optimal solution might utilize a subnetwork of the actual *KNC*. The best size of the subcube is given by Theorem 2.3.

The folding digit can be selected from either the odd range or the even range. Let n_{fo} and n_{fe} denote the number of folding digits from the odd and even ranges, respectively. A particular layer l is affected by odd (even) foldings only if $n_l > n_o - n_{fo}$ (or $n_l > n_e - n_{fe}$); that is, an odd (or even) layer l would be folded $max(n_l - n_o + n_{fo}, 0)$ ($max(n_l - n_e + n_{fe}, 0)$) times after n_{fo} odd foldings (n_{fe} even foldings). Folding different digits in a particular range might result in different processor utilization. Next, we show how to select the folding digit to maximize processor utilization.

Lemma 2.2: Folding along the most significant digit in a particular range (odd or even) maximizes processor utilization.

Proof: Without loss of generality, assume that the folding digit for the next iteration is from the odd range. If every BPG_l was exactly of the same size as *LVL* the proof would be trivial. It can be easily shown that folding along any odd digit would result in uniform distribution of *LVL* tasks among processors.

Assume that for an odd layer l, $n_l < n_o$. The assignment procedure assigns nodes of this layer to the first k^{n_l} nodes of the *LVL*. So, the remaining $k^{n_o} - k^{n_l}$ nodes have one less task assigned to each. Now, assume that the folding digit for the next iteration is from the odd range. Folding any of the first (least significant) n_l digits would result in k^{n_l-1} nodes with k tasks and $k^{n_o-1} - k^{n_l-1}$ nodes with no tasks from layer l. However, folding any of the $n_o - n_l$ most significant digits, results in k^{n_l} nodes with one task each and $k^{n_o-1} - k^{n_l}$ nodes with no tasks from layer l. Clearly, the latter folding results in a more uniform distribution of tasks among processors resulting in higher processor utilization (to the extent indicated by Theorem 2.3). According to Theorem 2.3, this is necessary to guarantee time-

optimal execution. Since n_l varies from one odd layer to another, in general, selecting the most significant digit always guarantees that the selected digit is among the n_o - n_l most significant digits. Thus, selecting the most significant digit guarantees best processor utilization and ensures time optimality since it guarantees the best possible uniform distribution of tasks. □

Next, we will derive the total execution time for each learning pass and will show how to select the folding digits properly to minimize the time. We assume that the computations of $f(x)$ and $f'(x)$ take $c_1 t_r$ and $c_2 t_r$ time, respectively, where f is the activation function, and c_1 and c_2 are constants.

Theorem 2.4: The time required by each learning pass for an L-layer *FFANN* (assuming L is an even integer) on the contracted *VKNC* after folding α even and β odd digits is

$$T^t(\alpha,\beta) = \sum_{l=1,\, odd\, l}^{L-1} O_l^{\beta}[\, m_1(E_{l+1}^{\alpha} + m_4 E_{l-1}^{\alpha}) + m_2 (n_{l-1}^{\alpha} + n_{l+1}^{\alpha}) + m_3 \,] + \sum_{l=0,\, even\, l}^{L} E_l^{\alpha}[\, m_1(m_4 O_{l+1}^{\beta} + O_{l-1}^{\beta}) + m_2 (n_{l-1}^{\beta} + n_{l+1}^{\beta}) + m_3 \,] \tag{2.15}$$

where $t = \alpha + \beta$, $m_1 = 2\, t_r$, $m_2 = (\lceil k/2 \rceil + 1)\, t_c + 2\, t_r$, $m_3 = c_1 + c_2$, and $m_4 = 2.5$.

Proof: We compute the overall computation time by determining the execution time required by each step involved in each learning pass. Without loss of generality, assume that k is even. The forward pass consists of three steps. $E_l^{\alpha} O_l^{\beta}$ multiplications and $E_l^{\alpha} O_l^{\beta}$ additions take place during Step 1, for layer l, requiring a total time of $2\, E_l^{\alpha} O_l^{\beta}\, t_r$. The $net_v^l(p)$ term is computed in Step 2 and broadcasted in Step 3 using the given fan-in and fan-out algorithms, respectively. Using the results of Theorem 3.3, we can show that these two steps take $O_l^{\beta} \lfloor k/2 \rfloor (n_{l-1}^{\alpha} (t_r + t_c) + n_{l+1}^{\alpha} t_c)$ time units for each odd layer l, and

$E_l^{\alpha} \lfloor k/2 \rfloor (n_{l-1}^{\beta} (t_r + t_c) + n_{l+1}^{\beta} t_c)$ units for each even layer l. By adding up the times required by different steps for each layer, we obtain equation (2.16)

$$T_{forward}(t) = \sum_{l=1 \text{ and odd } l}^{L-1} [\, 2O_l^{\beta} E_{l-1}^{\alpha} t_r + O_l^{\beta} \lfloor k/2 \rfloor (n_{l-1}^{\alpha} (t_r + t_c) + n_{l+1}^{\alpha} t_c) + O_l^{\beta} c_1 t_r \,] + \sum_{l=0 \text{ and even } l}^{L} [\, 2O_{l-1}^{\beta} E_l^{\alpha} t_r + E_l^{\alpha} \lfloor k/2 \rfloor (n_{l-1}^{\beta} (t_r + t_c) + n_{l+1}^{\beta} t_c) + E_l^{\alpha} c_1 t_r \,] \tag{2.16}$$

Using a similar approach, we obtain the time taken by the backward pass for each pattern as follows

$$T_{backward}(t) = \sum_{l=1 \text{ and odd } l}^{L-1} [\, 2O_l^{\beta} E_{l+1}^{\alpha} t_r + O_l^{\beta} \lfloor k/2 \rfloor (n_{l+1}^{\alpha} (t_r + t_c) + n_{l-1}^{\alpha} t_c) + O_l^{\beta} c_1 t_r \,] + \sum_{l=0 \text{ and even } l}^{L} [\, 2O_{l+1}^{\beta} E_l^{\alpha} t_r + E_l^{\alpha} \lfloor k/2 \rfloor (n_{l+1}^{\beta} (t_r + t_c) + n_{l-1}^{\beta} t_c) + E_l^{\alpha} c_1 t_r \,] \tag{2.17}$$

The weight increments are computed after each pattern presentation. The weights, however, might be updated after each pattern presentation or after presentation of all patterns (on-line or off-line training). The computation here is done for an on-line training. Computation of each weight increment involves one addition and one multiplication. Thus, for an even (odd) layer l a total of $2E_l^{\alpha} O_{l-1}^{\beta} t_r$ ($2E_{l-1}^{\alpha} O_l^{\beta} t_r$) steps are required to compute weight increments and an additional $E_l^{\alpha} O_{l-1}^{\beta} t_r$ ($E_{l-1}^{\alpha} O_l^{\beta} t_r$) steps are required to update the weights. Therefore, for each pattern presentation, the weight update takes

$$T_{weight\text{-}update}(t) = \sum_{l=1 \text{ and odd } l}^{L-1} 3O_l^{\beta} E_{l-1}^{\alpha} t_r + \sum_{l=2 \text{ and even } l}^{L} 3O_{l-1}^{\beta} E_l^{\alpha} t_r \tag{2.18}$$

time units. The total time required by each learning pass is given by equation (2.19).

$$T(t) = T_{forward}(t) + T_{backward}(t) + T_{weight\text{-}update}(t) \qquad (2.19)$$

After some simple algebraic simplifications, $T^t(\alpha,\beta)$ can be obtained and is as stated in the theorem. □

We can use Theorem 2.4 to determine the folding range (odd or even) which would minimize the execution time at each folding iteration. Assume that after t contractions (see definition 2.4) , α even and β odd digits have been used for folding. Let $T^{t+1}(\alpha,\beta+1)$ denote the total execution time on the *KNC* obtained after the $(t+1)^{st}$ folding when the folding digit at iteration $t+1$ is chosen from the odd range. Similarly, let $T^{t+1}(\alpha+1,\beta)$ denote the total execution time on the *KNC* obtained after the $(t+1)^{st}$ folding when the folding digit at iteration $t+1$ is chosen from the even range. Clearly, folding an odd digit at iteration $t+1$ results in lower overall time if $T^{t+1}(\alpha,\beta+1) < T^{t+1}(\alpha+1,\beta)$, and there exists an odd folding digit after t contractions. Based on the above, we have developed the folding algorithm listed below:

Algorithm 2.3:

/* Folding Algorithm *FA* */

Begin:

$\alpha = 0$; $\beta = 0$;

$n_f = n_v - n_a$

FOR ($i = 1$ to n_f) DO

IF ($T^{t+1}(\alpha, \beta+1) < T^{t+1}(\alpha+1, \beta)$ and $n_o > 0$)

/* Fold the most significant odd digit */

$n_o = n_o - 1;$

$\alpha = \alpha + 1;$

ELSE

/*Fold the most significant even digit */

$n_e = n_e - 1;$

$\beta = \beta + 1;$

ENDIF

ENDFOR

End

Next, we show that if *FA* is used for folding the *VKNC*, then the overall learning simulation will be optimal.

Theorem 2.5: The folding scheme described by Algorithm *FA* minimizes the total execution time of the learning phase on the resulting *KNC*.

Proof: We use mathematical induction in this proof. Let $P(j)$ represent the following statement: *Algorithm FA results in an optimal simulation after j foldings, for* $j \geq 1$.

Induction Basis: ($P(1)$) is true since by definition for one folding *FA* folds the digit which results in the lowest overall execution time.

Induction Step: We need to show that $P(j)$ implies $P(j+1)$.

Assume that there exists a folding algorithm *OPT* which results in optimal simulation of the *FFANN* on the *KNC*. We assume that after j foldings, *OPT* has folded α even and β odd digits, while *FA* has folded γ even and ϵ odd digits for the same *FFANN* and *VKNC*.

Clearly, $\alpha + \beta = \epsilon + \gamma = j$. Let $T^j_{opt}(\alpha,\beta)$ and $T^j_{FA}(\epsilon,\gamma)$ denote the overall execution times of a learning pass when the *OPT* and *FA* algorithms are implemented, respectively.

Using the above notation and Theorem 2.4 we can obtain $T^j_{opt}(\alpha,\beta)$ and $T^j_{FA}(\epsilon,\gamma)$ as follows

$$T^j_{opt}(\alpha,\beta) = \sum_{l=1,\ odd\ l}^{L-1} O_l^{\beta}[\, m_1(E^{\alpha}_{l+1} + m_4\, E^{\alpha}_{l-1}) + m_2\,(n^{\alpha}_{l-1} + n^{\alpha}_{l+1}) + m_3\,] + \sum_{l=0,\ even\ l}^{L} E_l^{\alpha}[\, m_1(\, m_4\, O^{\beta}_{l+1} + O^{\beta}_{l-1}) + m_2\,(n^{\beta}_{l-1} + n^{\beta}_{l+1}) + m_3\,] \tag{2.20}$$

$$T^j_{FA}(\epsilon,\gamma) = \sum_{l=1,\ odd\ l}^{L-1} O_l^{\gamma}[\, m_1(E^{\epsilon}_{l+1} + m_4\, E^{\epsilon}_{l-1}) + m_2\,(n^{\epsilon}_{l-1} + n^{\epsilon}_{l+1}) + m_3\,] + \sum_{l=0,\ even\ l}^{L} E_l^{\epsilon}[\, m_1(\, m_4\, O^{\gamma}_{l+1} + O^{\gamma}_{l-1}) + m_2\,(n^{\gamma}_{l-1} + n^{\gamma}_{l+1}) + m_3\,] \tag{2.21}$$

The induction hypothesis can be stated as follows: $T^j_{FA}(\epsilon,\gamma) = T^j_{opt}(\alpha,\beta)$. We need to prove that $T^{j+1}_{FA} = T^{j+1}_{opt}$. We can easily show that the equality holds if $\alpha = \gamma$ and $\beta = \epsilon$ by using the induction hypothesis and the definition of *FA*. It remains to prove the statement when $\alpha \neq \epsilon$ and $\beta \neq \gamma$.

We need to examine two possible cases *i*) $\alpha < \epsilon$, $\beta > \gamma$ and *ii*) $\alpha < \epsilon$, $\beta > \gamma$. We show that for both cases $T^{j+1}_{FA} = T^{j+1}_{opt}$.

Case I) $\alpha < \gamma$ and $\beta > \epsilon$:

Without loss of generality, assume that *OPT* folds an odd digit at iteration j+1. We show that, for this case, $T^{j+1}_{FA} = T^{j+1}_{opt}$ holds whether *FA* folds an odd or even digit.

Case I.1) FA folds an odd digit:

In this case we assume that both algorithms fold an odd digit. The execution time after $j + 1$ foldings based on the above assumption in iteration j+1 is given by (2.22).

$$T^{j}_{opt}(\alpha,\beta+1) = \sum_{l=1,\,odd\,l}^{L-1} O_l^{\beta+1}[\, m_1(E^{\alpha}_{l+1} + m_4 E^{\alpha}_{l-1}) + m_2 (n^{\alpha}_{l-1} + n^{\alpha}_{l+1}) + m_3] + \sum_{l=0,\,even\,l}^{L} E^{\alpha}_{l}[\, m_1(m_4 O^{\beta}_{l+1}+1 + O^{\beta+1}_{l-1}) + m_2 (n^{\beta+1}_{l-1} + n^{\beta+1}_{l+1}) + m_3] \tag{2.22}$$

and,

$$T^{j}_{FA}(\epsilon,\gamma+1) = \sum_{l=1,\,odd\,l}^{L-1} O_l^{\gamma+1}[\, m_1(E^{\epsilon}_{l+1} + m_4 E^{\epsilon}_{l-1}) + m_2 (n^{\epsilon}_{l-1} + n^{\epsilon}_{l+1}) + m_3] + \sum_{l=0,\,even\,l}^{L} E^{\epsilon}_{l}[\, m_1(m_4 O^{\gamma+1}_{l+1} + O^{\gamma+1}_{l-1}) + m_2 (n^{\gamma+1}_{l-1} + n^{\gamma+1}_{l+1}) + m_3] \tag{2.23}$$

We need to show that $T^{j+1}_{opt}(\alpha,\beta+1) - T^{j}_{opt}(\alpha,\beta) = T^{j+1}_{FA}(\epsilon,\gamma+1) - T^{j}_{FA}(\epsilon, \gamma)$. Let

$$T^{j+1}_{opt}(\alpha\,,\,\beta+1) - T^{j}_{opt}(\alpha\,,\,\beta) = \sum_{l=0}^{L} \Delta_l^{\beta+1} \tag{2.24}$$

and

$$T^{j+1}_{FA}(\epsilon\,,\,\gamma+1) - T^{j}_{FA}(\epsilon\,,\,\gamma) = \sum_{l=0}^{L} \Delta_l^{\gamma+1} \tag{2.25}$$

where $\Delta_l^{\beta+1}$ ($\Delta_l^{\gamma+1}$) represents the additional execution time required by layer l once an odd digit is folded by the *OPT* (*FA*) algorithm assuming that β (γ) odd digits have been folded previously. We show that for every layer l, $\Delta_l^{\gamma+1} = \Delta_l^{\beta+1}$.

The execution time required by layer l depends on how neurons in layers l-1, l, and l+1 are assigned to physical processors. Therefore, we have to consider how these layers are folded. There are at most five possible ways by which folding of an additional digit might affect execution time of layer l depending on how this folding affects layers l-1, l, and l+1. Note that each of these layers might or might not be folded at the most recent iteration. These cases are listed in Table 2.1.

Table 2.1: Possible foldings for a given layer.

	layer l-1	layer l	layer l+1
Case 1	Not Folded	Not Folded	Not Folded
Case 2	Not Folded	Not Folded	Folded
Case 3	Not Folded	Folded	Not Folded
Case 4	Folded	Not Folded	Not Folded
Case 5	Folded	Not Folded	Folded

We first show that for every odd layer l and for any of the above five cases $\Delta_l^{\gamma+1} = \Delta_l^{\beta+1}$.

Using (2.22) and (2.23) we can find $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1}$ for an odd layer as follows:

$$\begin{aligned}\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} = (O_l^{\beta+1} - O_l^{\beta})\,[\, m_1(E_{l+1}^{\alpha} + m_4 E_{l-1}^{\alpha}) + m_2 (n_{l-1}^{\alpha} + n_{l+1}^{\alpha}) + m_3 \,] - \\ (O_l^{\gamma+1} - O_l^{\gamma})\,[\, m_1(E_{l+1}^{\epsilon} + m_4 E_{l-1}^{\epsilon}) + m_2 (n_{l-1}^{\epsilon} + n_{l+1}^{\epsilon}) + m_3 \,]\end{aligned} \tag{2.26}$$

In cases 1,2, 4, and 5 layer l is not folded. So, $O_l^{\beta+1} - O_l^{\beta} = O_l^{\gamma+1} - O_l^{\gamma} = 0$. This clearly implies $\Delta_l^{\gamma+1} = \Delta_l^{\beta+1}$. In case 3, however, only layer l is folded. Since $\beta > \epsilon$, we have

$$m_3(O_l^{\beta+1} - O_l^{\beta}) \geq m_3 (O_l^{\gamma+1} - O_l^{\gamma}) \tag{2.27}$$

In addition, $n_{l-1}^{\alpha} \geq n_{l-1}^{\epsilon}$ and $n_{l+1}^{\alpha} \geq n_{l+1}^{\epsilon}$ because $\alpha < \gamma$. So,

$$m_1 (E_{l+1}^{\alpha} + m_4 E_{l-1}^{\alpha}) \geq m_1 (E_{l+1}^{\epsilon} + m_4 E_{l-1}^{\epsilon}) \tag{2.28}$$

Let $\gamma = \alpha + \theta$ and $\beta = \epsilon + \theta$, where $\theta > 0$. Since both algorithms fold layer l and $\beta > \epsilon$, we have

$$\begin{aligned}O_l^{\beta+1} - O_l^{\beta} &= k^{i+\theta}(k-1) \\ O_l^{\gamma+1} - O_l^{\gamma} &= k^{i}(k-1)\end{aligned} \tag{2.29}$$

for some $i \geq 0$. Since $\alpha < \gamma$, we have $E_{l+1}^{\alpha} \geq k^z$, $E_{l-1}^{\alpha} \geq k^z$, $E_{l+1}^{\epsilon} \leq k^{z+\theta}$, and $E_{l-1}^{\epsilon} \leq k^{z+\theta}$, where $z \geq 0$. Therefore, we can derive equation (2.30).

$$(O_l^{\beta+1} - O_l^{\beta})\,[\,m_1(E_{l+1}^{\alpha} + m_4\,E_{l-1}^{\alpha})] \;\geq\; (O_l^{\gamma+1} - O_l^{\gamma})\,[\,m_1(E_{l+1}^{\epsilon} + m_4\,E_{l-1}^{\epsilon})] \qquad (2.30)$$

By combining (2.27), (2.28), and (2.30) we observe that for case 2, $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} \geq 0$. Therefore, for any of the five possible cases and for an odd layer we have shown that $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} \geq 0$.

At this point, we must show that this also holds for every even layer. Using (2.22) and (2.23) we can find $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1}$ for an even layer as follows:

$$\begin{aligned}\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} = {} & E_l^{\alpha}[m_1((O_{l-1}^{\beta+1} - O_{l-1}^{\beta}) + m_4(O_{l+1}^{\beta+1} - O_{l+1}^{\beta})) + m_2\,((n_{l-1}^{\beta+1} - n_{l-1}^{\beta}) + (n_{l+1}^{\beta+1} - n_{l+1}^{\beta}))] - \\ & E_l^{\epsilon}[m_1(m_4\,(O_{l+1}^{\gamma+1} - O_{l+1}^{\gamma}) + (O_{l-1}^{\gamma+1} + O_{l-1}^{\gamma})) + m_2\,((n_{l-1}^{\gamma+1} - n_{l-1}^{\gamma}) + (n_{l+1}^{\gamma+1} - n_{l+1}^{\gamma}))]\end{aligned} \qquad (2.31)$$

None of the three layers is folded in case 1, so clearly $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} = 0$.

For case 2,

$$\begin{aligned}\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} = {} & E_l^{\alpha}[\,m_1\,(O_{l+1}^{\beta+1} - O_{l+1}^{\beta})) + m_2\,(n_{l+1}^{\beta+1} - n_{l+1}^{\beta})] - \\ & E_l^{\epsilon}[\,m_1(O_{l+1}^{\gamma+1} - O_{l+1}^{\gamma}) + m_2\,(n_{l+1}^{\gamma+1} - n_{l+1}^{\gamma})]\end{aligned} \qquad (2.32)$$

Similar to the rationale used for (3.25) we can show that

$$m_4\,m_1[\,E_l^{\alpha}\,(O_{l+1}^{\beta+1} - O_{l+1}^{\beta}) - E_l^{\epsilon}\,(O_{l+1}^{\gamma+1} - O_{l+1}^{\gamma})] \geq 0 \qquad (2.33)$$

In addition, since $n_{l+1}^{\gamma+1} - n_{l+1}^{\gamma} = -1$ we obtain the following:

$$m_2\,[E_l^{\alpha}(n_{l+1}^{\beta+1} - n_{l+1}^{\beta}) - E_l^{\epsilon}(n_{l+1}^{\gamma+1} - n_{l+1}^{\gamma})\,] = m_2(E_l^{\epsilon} - E_l^{\alpha}) \geq 0 \qquad (2.34)$$

Using the above results, we have $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} \geq 0$.

In case 3, by definition layers l-1 and l+1 are not folded. Layer l is not folded either since it is an even layer. So, the result is similar to that of case 1. The proof for case 4 can be obtained from that of case 2 by replacing l+1 by l-1. By combining results of cases 2 and 4 we observe that in case 5, $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} \geq 0$. Thus, $\Delta_l^{\beta+1} - \Delta_l^{\gamma+1} \geq 0$ holds for every layer of the *FFANN*. Based on this result we can derive equation (2.35).

$$T_{opt}^{j+1}(\alpha,\beta+1) - T_{opt}^{j}(\alpha,\beta) \geq T_{FA}^{j+1}(\epsilon,\gamma+1) - T_{FA}^{j}(\epsilon,\gamma) \quad (2.35)$$

However, by definition of the *OPT* algorithm, the above time measures can only be equal. For this to be true $\alpha = \gamma$ and $\beta = \epsilon$ must be true. In other words, *FA* and *OPT* must result in identical mappings.

Case I.2) FA folds an even digit:

We need to show that

$$T_{FA}^{j+1}(\epsilon+1,\gamma) = T_{opt}^{j+1}(\alpha,\beta+1) \quad (2.36)$$

If *FA* folds and even digit at iteration $j+1$, by definition

$$T_{FA}^{j+1}(\epsilon+1,\gamma) \leq T_{FA}^{j+1}(\epsilon,\gamma+1) \quad (2.37)$$

However, we have already shown that

$$T_{opt}^{j+1}(\alpha,\beta+1) = T_{FA}^{j+1}(\epsilon,\gamma+1) \quad (2.38)$$

Equations (2.37) and (2.38) clearly indicate that (2.36) holds.

Case II) $\alpha > \gamma$ and $\beta < \epsilon$:

Using a similar approach we can easily show that this case also leads to the fact that *OPT* and *FA* must result in identical mappings. □

2.3 Mapping Radial Basis Function Networks

In this section we introduce an efficient scheme for simulation of *Radial Basis Function* networks *RBF*'s on *KNC*'s. We consider both fully supervised and partially supervised algorithms for training *RBF*'s. We show that as far as mapping is concerned, fully supervised training of *RBF* networks is very similar to training a two-layer *FFANN*. The partially supervised training takes place in two phases. In the first phase, certain parameters related to the radial basis functions used in the network (namely the centers and

widths) are determined using an unsupervised scheme. We introduce an efficient parallel scheme for performing the unsupervised phase. The second phase is a supervised training algorithm similar to training a two-layer *FFANN*. Hence, we can utilize the scheme we already introduced to map the second phase of the partially supervised algorithm. Next, we briefly review the *RBF* network and then present our mapping scheme.

2.3.1 Radial Basis Function *ANN*'s

Radial Basis Function is an *ANN* model designed based on the "locally tuned" response observed in many parts of biological nervous systems such as cochlear stereocilia cells [33]. *RBF*'s have been used in a variety of applications including interpolations [53][68][12], probability density estimation [60][22][75], and multivariate function approximation [66] [33]. These models are particularly suitable for approximating continuous or piecewise continuous functions $f: R^n \rightarrow R^L$ where n is sufficiently small [33]. Next we describe the basic *RBF* network and its training algorithm. Our description is based on material in [33].

RBF's have a feedforward structure as shown in Figure 2.6. It consists of two layers of neurons, a hidden layer and an output layer. The model employs certain number (J) of hidden units , each receiving an n-dimensional (real-valued) input pattern. We represent the k^{th} input pattern by an n-dimensional vector $X^k = (x_1^k, x_2^k,...,x_n^k)$ and its corresponding output pattern by an L-dimensional vector $Y^k = (y_1^k, y_2^k,...,y_L^k)$. We denote the number of input/output training patterns by m. The hidden units are fully connected to the output units. We denote the output of hidden unit j by z_j.

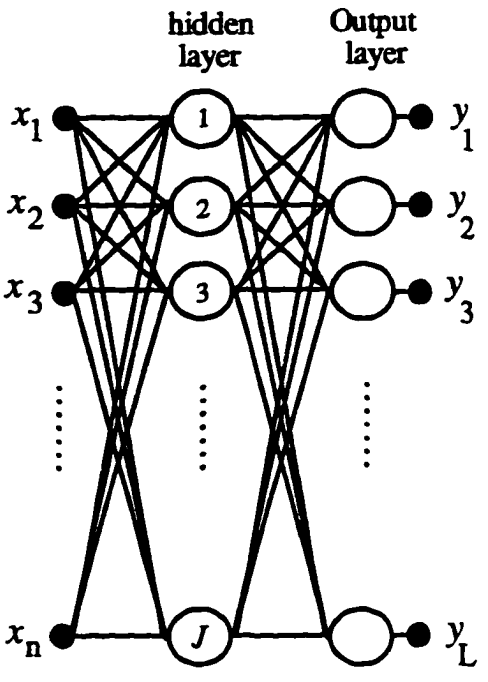

Figure 2.6: The *RBF* network.

Unlike models we have considered so far, *RBF*'s do not have synaptic weights associated with hidden layer units. These units do not compute their outputs based of weighted-sum of their inputs. Instead, each hidden unit computes its output (z_j) based on the "closeness" of the input pattern to an *n*-dimensional vector (μ_j) which is called *center*. Formally, the output of hidden unit *j* can be represented as follows:

$$z_j(X) = G\left(\frac{\| X - \mu_j \|}{\sigma_j^2}\right) \tag{2.39}$$

where *G*(.) is a strictly positive radially symmetric function with a unique maximum at its center μ_j which goes to zero rapidly away from the center. This function is also referred to as the *receptive filed* in the input space for hidden unit *j*. The parameter σ_j represents the width of this receptive field . In other words, *G*(.) has a significant value if distance $\| X - \mu_j \|$ is less than the width.

Examples of $G(.)$ are the Guassian and logistic functions:

$$\text{Guassian: } z_j(X) = e^{-\frac{\|X-\mu_j\|^2}{2\sigma_j^2}} \qquad \text{Logistic: } z_j(X) = \left[1 + e^{-\frac{\|X-\mu_j\|}{\sigma_j^2} - \theta_j}\right]^{-1} \quad (2.40)$$

where θ_j is an adjustable bias in the logistic function. The output of each output unit is computed in terms of the weighted sum of hidden-unit outputs. For instance, the l^{th} component of the output pattern for input pattern X is given by:

$$y_l = \sum_{j=1}^{J} w_{lj}\, z_j(X) \quad (2.41)$$

Training of an *RBF* basically corresponds to determining *free* parameters of the network for a given set of m training pairs. These parameters are the centers and the widths of the hidden-unit receptive fields (μ_j and σ_j) as well as output-layer weights (w_{ij}'s). Several training strategies are described in [33] including a fully supervised gradient-decent method [54][66] and several partially supervised schemes [55]. Next we describe each training algorithm in detail and explore its parallel implementation on k-ary n-cubes.

2.3.2 Fully Supervised Training of *RBF* networks

In the fully supervised training, the free parameters of the *RBF* are updated to minimize an error function E. Several studies [54][66] have considered the gradient-decent method over E to update the free parameters. Formally, the training takes place using the following updates

$$\Delta w_{ij} = \rho_w \frac{\partial E}{\partial w_{ij}} \quad , \quad \Delta\sigma_j = -\rho_\sigma \frac{\partial E}{\partial \sigma_j} \quad , \quad \Delta\mu_j = -\rho_\mu \nabla_{\mu_j} E \quad (2.42)$$

where ρ_μ , ρ_σ , and ρ_w are small positive constants.

If the delta rule is used for updating the output-layer weights, weight updates can be computed as follows:

$$\Delta w_{lj} = \rho_w (d_l - y_l) f'(net_l) z_j \tag{2.43}$$

where $net_l = \sum_{j=1}^{J} w_{lj} z_j$, and d_l is the l^{th} component of the actual output. In the fully supervised training, the hidden layer updates are computed in terms of output-layer errors. Clearly, this scheme updates receptive field centers and widths by back propagating the output error through the network.

Clearly the computations involved in fully supervised training of *RBF* networks are very similar to those of a feedforward neural network trained with the backpropagation algorithm. In fact, an *RBF* network with logistic basis function is very similar to a two-layer feedforward network [33]. The only difference is in the computation of hidden-layer units. As far as mapping is concerned, such differences are relatively insignificant. Hence, the mapping of a fully supervised *RBF* network can be performed in the same exact way a two-layer feedforward neural network would be mapped using the scheme described in Sections 2.1 and 2.2.

2.3.3 Partially Supervised Training of *RBF*

The fully supervised scheme leads to training time similar to that of sigmoidal-type networks [85]. The slow convergence is mainly due to inefficient use of locally tuned hidden units [33]. Hassoun [33] describes several training schemes which "decouple" learning at the hidden layer from the output layer errors for *RBF*'s. Essentially, the computations of hidden-layer receptive field parameters are performed independent of output-layer errors. Several schemes for computing receptive field centers and widths are described in [33].

Moody and Darken [55] proposed an unsupervised training scheme for locating centers of receptive fields which require relatively few *RBF*'s [33]. This scheme leads to a very efficient representation of data because of the small number of receptive fields used. The algorithm proceeds as follows. Originally, a number of training patterns are selected at random as centers of receptive fields. Each receptive field represents a region or class of the input space. The remaining training patterns are then assigned to the class with the closest center. In other words, pattern X_i is assigned to class j if center μ_j is closer to X_i than any other center. Next, the center of each class is recomputed as the average of all patterns assigned to the class. This process is repeated until all centers remain unchanged. We formally represent the above algorithm as follows:

Algorithm 2.4:

/* Batch-mode center (μ) search */

Begin:

For (class j = 1 to J) Do

Select a random input pattern as the center ;

While (centers μ_j's change) Do

1. For (i = 1 to m - k) Do

1.1 Initialize $minimum_i$ to a large value ;

1.2 For (j = 1 to J) Do

1.2.1 Compute $\| \mu_j - X_i \|$;

1.2.2 If ($\| \mu_j - X_i \| < minimum_i$)

$minimum_i = \| \mu_j - X_i \|$;

Class (X_i) = J

EndIf

EndFor

EndFor

2. For (j = 1 to J) Do

2.1 $$\mu_j = \sum_{X_i \in \text{Class } j} X_i / \| \text{class } j \|$$

EndFor

EndWhile

End

Next, we explore parallel implementation of the algorithm . The training algorithm proceeds in a well defined and uniform manner. It consists of two major steps. During the first step, each pattern is involved in identical computations. Basically, the pattern is assigned to the class with the closest center. Clearly, patterns can be processed independently and concurrently. In the second step, new centers are calculated for each class as the average of all patterns assigned to the class in the first step. These computations can take place for each class independently. However, the number of patterns assigned to each class might be different. Notice that in practice centers are gradually adjusted to represent densely populated regions in the input space. Clearly, the number of patterns in each region may not be identical.

There are various ways to exploit parallelism in the above algorithm. We propose a parallel implementation of the algorithm based on partitioning the training set. This way, we can fully exploit the parallelism involved in the first step of the algorithm while executing

the second step with reasonable efficiency as we will explain. The key here issue is how to partition the training set among physical processors of the *KNC* architecture. We show that a uniform distribution of patterns leads to efficient simulation of the training algorithm. We make the following assumptions in subsequent derivations:

- $\| \mu_j - X_i \|$ can be computed on a single processor in $c_1 n t_r$ time units, where c_1 is a positive constant, n is the dimensionality of the input pattern, and t_r is the computation time for a simple arithmetic operation.
- Q_p patterns are assigned to processor p of the s-dimensional *KNC*, such that at least one pattern is assigned to each processor; i.e., $1 \leq Q_p \leq m - k^s + 1$.

We introduce a parallel version of Algorithm 2.4 which efficiently computes the centers of receptive fields. Our approach is based on partitioning the training set and adheres to the above assumptions. The proposed algorithm is listed below.

Algorithm 2.5:

Begin:

While (centers are changing) Do

1. For (all processors p of the s-dimensional *KNC*) Parallel Do

1.1 For (all patterns X_i assigned to processor p) Do

1.1.1 Initialize *minimum* to a very large value ;

1.1.2 For (each hidden unit j) Do

1.1.2.1 compute $\| \mu_j - X_i \|$;

1.1.2.2 If ($\| \mu_j - X_i \| < minimum$)

$minimum = \| \mu_j - X_i \|$;

Class(X_i) = j ;

EndIf

EndFor

EndFor

1.2 For (all hidden units j = 1 to J) Do

For (each pattern X_i in Class j) Do

sum(j) = *sum*(j) + 1;

EndFor

μ_{jp} = *sum(j) / size-of-*class j ;

EndFor

EndFor

2. For (all hidden units j = 1 to J) Do

For (all processors p) Do

sum(μ_j)= sum(μ_j)+ μ_{jp} ;

EndFor

μ_j = sum(μ_j) / k^2 ;

EndFor

Endwhile

End;

Next, we determine the iteration time of Algorithm 2.5. Iteration time is the execution time of one iteration of the training algorithm.

Theorem 2.6: One iteration of Algorithm 2.4 on an s-dimensional *KNC* takes

$$T(s) = \max(Q_p)(c_1 n t_r + 6t_r) + Js(\lceil k/2 \rceil t_c + 2t_r) \quad (2.44)$$

time units.

Proof: We assume that the Step 1.1.1 takes negligible time. We have assumed that $| \mu_j - X_i |$ takes $c_1 n t_r$ time units. Steps 1.1.2.1 and 1.1.2.2 consist of 4 simple arithmetic operations. Hence, they require $4t_r$ time units. Therefore Step 1.1 on processor p takes $Q_p(c_1 n + 4)t_r$ time units. The worst case for Step 1.2 happens when all patterns in at least one processor belong to the same class. In this case, Step 1.2 takes $2\max(Q_p)t_r$. In Step 2, the overall average for each of the J centers is calculated. According to the results of Theorem 2.1 this step takes $Js(\lceil k/2 \rceil + 2t_r)$. Hence, one iteration of the algorithm takes at most

$$\max(Q_p)(c_1 n t_r + 6t_r) + Js(\lceil k/2 \rceil t_c + 2t_r) \quad (2.45)$$

time units. □

Theorem 2.7: For a given s, a uniform distribution of m patterns among k^s processors minimizes the iteration time of Algorithm 2.5, $T(s)$.

Proof: Clearly, a uniform distribution minimizes the first term of $T(s)$. The second term on the other hand is independent of pattern distribution. Hence, a uniform distribution leads to the minimum value of $T(s)$ for a given s. □

For a uniform distribution, $T(s)$ (2.44) becomes:

$$T(s) = \lceil mk^{-s} \rceil (c_1 n t_r + 6t_r) + Js(\lceil k/2 \rceil t_c + 2t_r) \quad (2.46)$$

Theorem 2.8: The minimum of function $T(s)$ with respect to s occurs at:

$$n_{opt} = \log_k \frac{m(c_1 n + 6)\ln k}{J(\lceil k/2 \rceil t_c + 2t_r)} \tag{2.47}$$

Proof: $T(s)$ is a convex function of s since its second derivative with respect to s is always positive. For instance, when m is divisible by k^s the second derivative of the function is given by:

$$\frac{\partial^2 T(s)}{\partial s^2} = m \ln k \, k^{-s} t_r (nc_1 + 6) \tag{2.48}$$

which is always positive. A similar result can be obtained if m is not divisible by k^s. The critical point of the function is obtained by setting the first derivative equal to zero. This leads to (2.47). Notice that since the function is convex, its extreme point is a minimum.□

Now, assume that the actual parallel architecture is an n_a-dimensional *KNC*. Using Theorem 2.8, we can determine the dimensionality of the subcube of the *KNC* which executes Algorithm 2.5 optimally (in minimum time). Notice that we are looking for an integer value s_{opt} , where $0 \le s_{opt} \le n_a$. (2.47) gives the real-valued minimum of the function. However, from basic calculus [9] the minimum of the convex function $T(s)$ (with respect to s) in the closed interval $[0,n_a]$ is either the critical point of the function or one of the end points of the interval. Hence, the dimensionality of the subcube of the n_a-dimensional *KNC* which executes Algorithm 2.5 in minimum time is given by (2.49).

$$s_{opt} = \begin{cases} \lceil n_{opt} \rceil & \text{if } 0 \le n_{opt} \le n_a \text{ and } T(\lceil n_{opt} \rceil) < T(\lfloor n_{opt} \rfloor) \\ \lfloor n_{opt} \rfloor & \text{if } 0 \le n_{opt} \le n_a \text{ and } T(\lceil n_{opt} \rceil) \ge T(\lfloor n_{opt} \rfloor) \\ 0 & \text{if } n_{opt} \le 0 \\ n_a & \text{if } n_{opt} > n_a \end{cases} \tag{2.49}$$

We can use (2.49) to implement Algorithm 2.5 on a *KNC* architecture optimally. Once centers of the receptive fields are computed using Algorithm 2.5, we can proceed with the computation of the output-layer weights. These weights can be computed using the delta rule [33] which is similar to finding weights for a two-layer feedforward network. Hence, the mapping of the second phase of the partially supervised *RBF* network can be performed using the scheme described in Sections 2.1 and 2.2.

CHAPTER 3

MAPPING UNIT ALLOCATING NETWORKS

In this chapter we consider mapping of several unit allocating networks. A unit allocating neural network is one whose topology is modified during the training. This class of neural networks includes several important *ANN* models such as the cascade correlation learning algorithm [25] and adaptive resonance theory models [31]. The common feature of these models is the dynamic nature of their architecture which grows during the training. Hardware implementation of these models are impractical due to their dynamic nature. We develop a general mapping scheme for highly efficient parallel simulation of such networks.

We first consider the mapping of the *Cascade Correlation learning algorithm.* Cascade correlation [25] is an efficient supervised learning technique for neural networks. The learning algorithm incrementally adds and trains hidden units to a minimal topology until a desired error bound is reached. The significant attributes of such a "unit-allocating" network are fast learning (with polynomial time complexity) and compact representation of data [33]. The resulting architecture is a multi-layer network with cascaded single-unit hidden layers. VLSI implementation of this structure is difficult due to its irregular connections and unbounded fan-in [25].

In Sections 3.1 and 3.2 we present a formal methodology for efficient parallel implementation of the Cascade Correlation algorithm on k-ary n-cubes (*KNC*'s). We develop a computational model which captures the inherent parallelism of output-unit and hidden-unit training phases of the algorithm. Moreover, our model allows pipelining of

several training patterns in order to further improve the efficiency of the implementation. The model we develop can easily be adapted to various parallel topologies. The mapping is done in two phases. The computational model is first mapped onto a virtual *KNC* of compatible size denoted by *VKNC*. Then, the *VKNC* is folded until a certain metric is optimized for a network with a certain number of hidden units, and the resulting size is less than or equal to the size of the actual *KNC*. In the Cascade Correlation algorithm the number of hidden units is not known in advance. To efficiently map the training of such a dynamic network, we consider an upper bound on the number of hidden units denoted by *Hmax*. We consider two optimization criteria defined based on 1) the execution time of the algorithm for a network with *Hmax* hidden units, and 2) the sum of execution times of the algorithm for all instances of the network with 0 through *Hmax* hidden units.

We propose efficient analytical schemes for mapping based on each criterion. We use the parameters for the benchmark application NETTalk [64] to evaluate the performance of our mappings. Experimental results show that our approach leads to near-optimal results for networks with H hidden units where, $H \leq Hmax$. In addition, we show that the proposed scheme leads to very efficient simulation of the training algorithm even if the number of hidden units exceeds *Hmax*. We also examine the effect of *Hmax* choice on the mapping. The minimization of each metric (assuming *Hmax* hidden units) has computational complexity $O(\log_k(L + Hmax))$, for a network with L output units. Based on the proposed mapping, task assignments for networks with 0 through *Hmax* hidden units are known apriori. Hence, no data transfer or task rescheduling is needed as the number of hidden units grows.

In Section 3.3 we consider the mapping of a popular clustering network called Adaptive Resonance Theory (*ART*) [31]. We show that the mapping of this algorithm is very similar to that of the Cascade Correlation training algorithm. We provide simulation results of efficient mapping of a network to implement the benchmark example for this case, as well.

3.1 Preliminaries

One of the most desirable attributes of a neural network learning algorithm is its efficiency. However, most learning algorithms have exponential time complexity [33]. This is particularly true about training multilayer neural networks with fixed topologies [33]. On the other hand, unit allocating networks [33] which allocate new units as needed have polynomial time complexity. Fahlman and Lebiere [25] introduced an efficient and practical unit-allocating learning technique called *Cascade Correlation.*

Cascade Correlation (*CC*) learning is a fast and efficient algorithm for supervised training of artificial neural networks [25],[33]. For brevity, we shall refer to the cascade correlation algorithm as the *CC* algorithm and to the resulting architecture as the *CC* architecture. The algorithm constructs a layered network by allocating hidden units one at a time until a desired error bound is reached. Each new hidden unit is fully connected to the input units and to any preexisting hidden units. Moreover, hidden units are fully connected to output units. Unlike conventional training algorithms, such as backpropagation, which train networks with wide and fixed hidden layers, the *CC* algorithm builds a deep network of cascaded units (see Figure 3.1). Each new hidden unit is trained to optimize a performance measure. Once trained, its weights are frozen for the

remainder of the training phase. This is an attempt to solve the *moving-target* problem [25] attributed to constantly changing weights of existing training algorithms. The scheme also eliminates the back-propagation of error signals through the network. Fahlman and Lebiere [25] have shown that the *CC* algorithm is much faster than the back-propagation algorithm.

The cascaded structure built by the *CC* training algorithm is not a good candidate for VLSI implementations due to its irregularity and unbounded fan-in [64]. In addition, the depth of the resulting structure could lead to long propagation delay since the delay is directly proportional to the number of hidden units allocated [64]. Notice that each allocated hidden unit serves as one network layer. Phatak and Koren [64] introduced a modified version of the *CC* algorithm which is intended to generate networks with small depth and restricted fan-in by controlling the connectivity. Their scheme generates a "strictly layered" network in which there are no connections that skip a layer. Their results reveal that imposing such restrictions leads to longer training time but results in a structure better suited for VLSI implementations. Although the *CC* algorithm cannot be easily implemented in a VLSI context, it can be efficiently implemented on parallel architectures as we shall demonstrate.

In this chapter, we investigate the problem of efficiently implementing the *CC* learning algorithm on existing parallel architectures. To our knowledge this has not been previously attempted which could be due to the dynamic nature of the architecture produced by the *CC* learning algorithm. We develop necessary computational models to capture the inherent parallelism of the algorithm. The models can be adapted to different parallel architectures. Here, we propose a mapping methodology for efficient simulation of the

algorithm on k-ary n-cubes. Our scheme does not impose any restrictions on the original training algorithm or network connectivity. The proposed scheme achieves efficiency by utilizing the inherent parallelism of the training procedure and by pipelining training patterns.

The mapping involves implementation of a computational model developed for the pipelined version of the Cascade Correlation algorithm (called *PCC* henceforth) onto an n_a-dimensional *KNC*. This consists of two steps. First, the *PCC* model is mapped onto a virtual *KNC* (called *VKNC*) of compatible size. Since the number of hidden units is not known apriori, we consider an upper bound for this value (called *Hmax*) and use a *PCC* model developed based on a network with *Hmax* hidden units. The size of the *VKNC* is then reduced until it matches that of the actual *KNC*. This process is referred to as *folding*. Further foldings maybe necessary if mapping onto a subcube of the *KNC* architecture leads to a more efficient simulation of the algorithm. The folding procedure is performed such that a desired metric for a network with *Hmax* hidden units is minimized. Since the actual size of the neural network is determined during the training, we consider two optimization criteria based on the size of the largest network (one with *Hmax* hidden units). One metric is the iteration time of the largest network and the other is the sum of iteration times of all possible network sizes less than or equal to *Hmax*. *Iteration time* refers to the execution time of one iteration of the algorithm. We show that a minimization approach based on either of these two metrics leads to efficient mapping of other instances of the network; that is to say, networks with *Hmax* hidden units are optimal whereas networks with less than *Hmax* units are near optimal. We also show that optimizing the sum of iteration times of

all instances of the network (assuming a maximum of *Hmax* hidden units) leads to a more efficient mapping of other network instances. Further, we show that our mapping approach leads to a very efficient simulation of the training algorithm even if the number of hidden units exceeds *Hmax*. In addition, we show that the choice of *Hmax* is not critical (within certain limits) if the sum of iteration times is used as criterion. Based on our approach, no task reassignment or data migration is needed on the *k*-ary *n*-cube as the number of hidden units grows during the training. We show that minimizing each metric for a network with *Hmax* hidden units has time complexity $O\left(\log_k(L + Hmax)\right)$, where L is the size of the output layer. This search can be performed off line without adding any computational overhead to the training.

3.1.1 The Cascade Correlation Learning Algorithm

We now introduce the notation used in this section. Figure 3.1 shows a *CC* network after H hidden units have been allocated. The external input is an N-dimensional vector $X = \{x_1, x_2, ..., x_N\}$, and the output is an L-dimensional vector $Y = \{y_1, y_2, ..., y_L\}$. The output of hidden unit i is denoted by z_i. The weight between input unit j and output unit i is denoted by w_{ij} for $1 \le i \le L$ and $1 \le j \le N$. p_{ri} (q_{jr}) denotes the weight between hidden unit r and input unit i (output unit j), where $1 \le i \le N$ ($1 \le j \le L$). The weight of the link joining hidden units i and j is denoted by v_{ji}. We represent the number of patterns in the training set by m.

CC learning consists of two phases: output-unit training and hidden unit training. The training begins with no hidden units. Input and output units are fully connected. Initially during the first phase, input-unit to output-unit weights (w_{ij}'s) are adjusted to

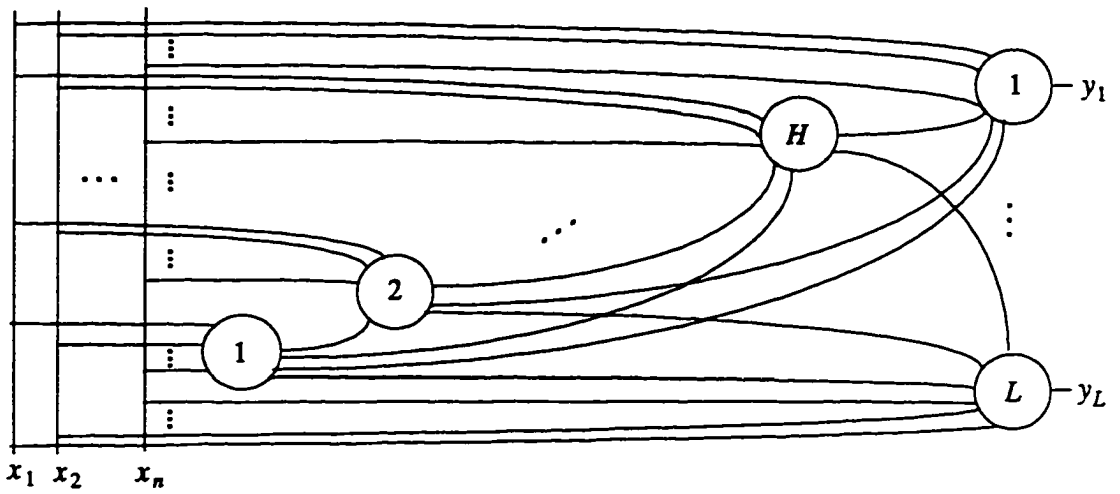

Figure 3.1: The cascade correlation architecture with H hidden units.

minimize an error measure (typically the sum of squared error). If the error remains above a predefined threshold after a certain number of training cycles, residual errors (E_l^k's) are recorded, where E_l^k is the difference between the actual (y_l^k) and desired (d_l^k) outputs for output-unit l when pattern k is presented.

During hidden-unit training a new unit is added to the network. This unit is fully connected to all input units and any preexisting hidden unit(s). Its outputs are not connected in this phase. Its incoming weights (p_{ri}'s and v_{ri}'s for hidden unit r) are then determined to maximize the covariance between its outputs and the residual errors computed during the latest output-unit training phase. For the r^{th} hidden unit this measure is formally represented as follows:

$$S_r = \sum_{l=1}^{L} \left| \sum_{k=1}^{m} (z_r^k - \overline{z}_r)(E_l^k - \overline{E}_l) \right| \tag{3.1}$$

where $\overline{z}_r$ and $\overline{E}_l$ are average values taken over all patterns [25]. Generally, several randomly initialized candidate hidden units are trained in parallel during this phase. Then,

the candidate unit which best maximizes the covariance measure S_r is chosen. The objective is to tune the new unit to the features not yet captured by the existing network [33]. The incoming weight will remain fixed during the subsequent training phases. At this point, the output of the new unit is connected to the output units, and its outgoing weights would be determined during the next output-unit training phase.

The two training phases are repeated, and new hidden units are allocated until the desired error bound is achieved. The delta learning rule is used for output-unit training. For hidden units on the other hand, a *gradient-ascent* optimization is performed to maximize the covariance measure S [25]. Incoming weights are adjusted to improve $\partial S/\partial p_{ri}$ (or $\partial S / \partial v_{ri}$). We denote the output layer (hidden layer) activation function by $F(.)$ ($G(.)$), the learning rate by ρ, and the sign of correlation between the candidate unit's output and that of output unit l by σ_l. The training phases of the algorithm are listed below.

Algorithm 3.1

/* Output-Unit Training */

Begin:

For (certain number of training cycles) Do

For (training patterns $k = 1$ to m) Do

For (hidden units $r = 1$ to H) Do

For (input units $i = 1$ to N) Parallel Do

Compute $p_{ri} x_i^k$

EndFor

For (hidden units $j = 1$ to r-1) Parallel Do

Compute $v_{rj} z_j^k$

EndFor

Compute $net_r^k = \sum_{i=1}^{N} p_{ri} x_i^k + \sum_{i=1}^{r-1} v_{ri} z_i^k$

Compute $z_r^k = G(net_r^k)$

EndFor

For (output units $j = 1$ to L) Parallel Do

For (input units $i = 1$ to N) Parallel Do

Compute $w_{ji} x_i^k$

EndFor

For (hidden units $r = 1$ to H) Parallel Do

Compute $q_{jr} z_r^k$

EndFor

Compute $net_j^k = \sum_{i=1}^{N} w_{ji} x_i^k + \sum_{i=1}^{H} q_{ji} z_i^k$

Compute $y_j^k = F(net_j^k)$

Compute $\Delta w_{ji}^k = \rho [d_j^k - F(net_j^k)] \frac{\partial F(net_j^k)}{\partial x_i^k} x_i^k$

Compute $\Delta q_{ji}^k = \rho [d_j^k - F(net_j^k)] \frac{\partial F(net_j^k)}{\partial z_i^k} z_i^k$

EndFor

EndFor

EndFor

If (error above the desired threshold) Do

For (l = 1 to L) Do

For(k = 1 to m) Do

$E_l^k = y_l^k - d_l^k$

EndFor

EndFor

End

Algorithm 3.2

/* Hidden-Unit training */

Begin

For (several candidate hidden units) Parallel Do

For (training patterns k = 1 to m) Do

For (hidden units r = 1 to H) Do

For (input units i = 1 to N) Parallel Do

Compute $p_{ri} x_i^k$

EndFor

For (hidden units j = 1 to r-1) Parallel Do

Compute $v_{rj} z_j^k$

EndFor

Compute $net_r^k = \sum_{i=1}^{N} p_{ri} x_i^k + \sum_{i=1}^{r-1} v_{ri} z_i^k$

Compute $z_r^k = G(net_r^k)$

EndFor

EndFor

For (input units $i = 1$ to N) Parallel Do

$$\Delta p_{H+1\,i} = \rho \sum_{l=1}^{L} \sum_{k=1}^{m} \left[\sigma_l (E_l^k - \overline{E}_l) \frac{\partial G(net_i^k)}{\partial x_i^k} x_i^k \right]$$

EndFor

For (hidden units $r = 1$ to H) Parallel Do

$$\Delta v_{H+1\,r} = \rho \sum_{l=1}^{L} \sum_{k=1}^{m} \left[\sigma_l (E_l^k - \overline{E}_l) \frac{\partial G(net_r^k)}{\partial z_r^k} z_r^k \right]$$

EndFor

EndFor

End

3.2 Parallel Implementation

3.2.1 The computational model

Both output-unit and hidden-unit phases of the training algorithm proceed in a well defined manner. The computations in each phase are flow dependent. We model the operation in each phase by a task graph. Nodes in this graph correspond to computations of units in the *CC* model and edges represent data communications between adjacent units. Due to the cascaded nature of the network, outputs of hidden units for a given training

pattern are calculated at different times. Each hidden unit can compute its output only after all the preceding hidden units have computed their outputs for the given pattern. Processing one pattern at a time with this flow dependency will certainly lead to poor performance. Here, we develop a pipelined version of the training algorithm to reduce the overall execution time by processing more than one input pattern in parallel. In the pipelined version, hidden and output units compute products associated with different patterns in a given cycle (one computation per neuron per cycle).

To illustrate pipelining of patterns we use the following example. Consider a *CC* network with 2 input units, 4 hidden units, and 2 output units. Assume that the network is to be trained with 6 training patterns (denoted by p_1 through p_6). Table 3.1 shows one pass of the training set through the network. Each row in this table represents a stage of the pipeline whereas each column represents a time step. We have assigned one stage to each output-layer unit and to each hidden-layer unit. At each time step a unit is busy with computation of the pattern listed in the corresponding stage. For instance, in the first time step hidden unit z_1 and output units y_1 - y_2 are computing products associated with pattern p_1. During the next 3 time steps, hidden units z_2 - z_4 compute tasks corresponding to $p1$,...,etc.

In general, when pattern k is presented, hidden unit r computes the products $p_{ri}x_i^k$ (for all $1 \le i \le N$) corresponding to the weighted inputs from the input units and the products $v_{rj}z_j^{kj}$ (for all $1 \le j \le r$-1 and $r \le k$) corresponding to the weighted inputs from the preceding hidden units and stores them. An output unit, say unit l, on the other hand computes products $w_{li}x_i^k$ (for all $1 \le i \le N$) corresponding to the weighted inputs from the

Table 3.1: Timing pattern for a pipelined *CC* network.

	Time								
Stages	1	2	3	4	5	6	7	8	9
z_1	p_1	p_2	p_3	p_4	p_5	p_6			
z_2		p_1	p_2	p_3	p_4	p_5	p_6		
z_3			p_1	p_2	p_3	p_4	p_5	p_6	
z_4				p_1	p_2	p_3	p_4	p_5	p_6
y_1	p_1	p_2	p_3	p_4	p_5	p_6			
y_2	p_1	p_2	p_3	p_4	p_5	p_6			

input units and products $v_{ij}z_j^{k-H}$ $(1 \le j \le H)$ corresponding to the weighted inputs from the allocated hidden units. The output of each hidden and each output unit for pattern k can be determined once all the product terms it needs have been computed.

Our ability to pipeline patterns depends on the ability of processors (of the parallel architecture) to store the computed products until the products can be added to the other products associated with the same pattern. This implies that the space complexity at each processor is increased by a factor of H with respect to a non-pipelined implementation. Fortunately, with declining memory cost and increased integration level[1] this increase in space complexity does not pose any practical problems. Notice that the number of training patterns which can be processed in parallel is equal to the number of allocated hidden units in the network. The algorithms for pipelined output-unit and hidden-unit training phases are listed next.

[1] DRAM's with one giga bits have already been built.

Algorithm 3.3:

/* Pipelined Output-Unit Training: */

Begin:

1. For (certain number of training cycles) Do

For ($k = 1$ to $m + H$) Do

For (hidden units and output units) Parallel Do

1.1 For (hidden units $r = 1$ to H) Parallel Do

1.1.1 For (input units $i = 1$ to N and hidden units $j = 1$ to $r - 1$) Parallel Do

Compute $p_{ri} x_i^k$ and $v_{rj} z_j^{k-j}$

EndFor

1.1.2 Compute $net_r^{k-r} = \sum_{i=1}^{N} p_{ri} x_i^{k-r} + \sum_{i=1}^{r-1} v_{ri} z_i^{k-r}$

1.2.3 Compute $z_r^{k-r} = G(net_r^{k-r})$

EndFor

1.2 For (output units $l = 1$ to L) Parallel Do

1.2.1 For (input units $i = 1$ to N and hidden units $j = 1$ to H) Parallel Do

Compute $w_{lj} x_i^k$ and $q_{lj} z_j^{k-H}$

EndFor

1.2.2 Compute: $net_j^{k-H} = \sum_{i=1}^{N} w_{ji} x_i^{k-H} + \sum_{i=1}^{H} q_{ji} z_i^{k-H}$

1.2.3 Compute: $y_j^{k-H} = F(net_j^{k-H})$

1.2.4 Compute in parallel : $\Delta w_{ji}^{k-H} = \rho\,[d_j^{k-H} - F(net_j^{k-H})]\,\dfrac{\partial F(net_j^{k-H})}{\partial x_i^{k-H}}\,x_i^{k-H}$

$$\Delta q_{ji}^{k-H} = \rho\,[d_j^{k-H} - F(net_j^{k-H})]\,\frac{\partial F(net_j^{k-H})}{\partial z_i^{k-H}}\,z_i^{k-H}$$

EndFor

EndFor

EndFor

EndFor

2. If (error above the desired threshold) Do

For ($l = 1$ to L) Parallel Do

2.1 For ($k = 1$ to m) Parallel Do

$E_l^k = y_l^k - d_l^k$

EndFor

2.2 Compute $\overline{E_l}$

EndFor

EndIf

End

Algorithm 3.4:

/* Pipelined Hidden-Unit training */

Begin:

For (several candidate hidden units) Parallel Do

1. For (training patterns $k = 1$ to m) Do

For (hidden units $r = 1$ to H) Parallel Do

1.1 For (input units $i = 1$ to N and hidden units $j = 1$ to r-1) Parallel Do

Compute $p_{ri} x_i^{k}$ and $v_{rj} z_j^{k-r}$

EndFor

1.2 Compute $net_r^{k-r} = \sum_{i=1}^{N} p_{ri} x_i^{k-r} + \sum_{i=1}^{r-1} v_{ri} z_i^{k-r}$

1.3 Compute $z_r^{k-r} = G(net_r^{k-r})$

EndFor

EndFor

2. For (training patterns $k = 1$ to m) Do

For (input units $i = 1$ to N and hidden units $r = 1$ to H) Parallel Do

2.1 $$\Delta p_{H+1\,i} = \Delta p_{H+1\,i} + \rho \sum_{l=1}^{L} \left[\sigma_l (E_l^{k-r} - \overline{E}_l) \frac{\partial G(net_r^{k-r})}{\partial x_i^{k-r}} x_i^{k-r} \right]$$

2.3 $$\Delta v_{H+1\,r} = \Delta v_{H+1\,r} + \rho \sum_{l=1}^{L} \left[\sigma_l (E_l^{k-r} - \overline{E}_l) \frac{\partial G(net_r^{k-r})}{\partial z_r^{k-r}} z_r^{k-r} \right]$$

EndFor

EndFor

EndFor

End

We model the pipelined learning architecture by a virtual two-layer network called pipelined cascade correlation network or *PCC*, for short. The *PCC* consists of a virtual input layer and a virtual output layer. The virtual input layer includes the input units as

well as the hidden units. The virtual output layer on the other hand consists of the hidden units and the output units. Hidden units are included in both virtual layers since each hidden unit receives external inputs as well as the outputs of any preceding hidden units and sends its output to output units as well as to any succeeding hidden units. Figure 3.2 illustrates the *PCC* for a network with H hidden units. The hidden units in the virtual input and output layers are labeled $z_{i1}, z_{i2}, ..., z_{iH}$ and $z_{o1}, z_{o2}, ..., z_{oH}$, respectively.

The *PCC* is intended to model the parallelism inherent to the pipelined *CC* algorithm. Edges of this model represent the communications which can take place in parallel between input-layer and output-layer units. Output-layer units on the other hand symbolize the concurrent computations. Each output-layer node represents a collection of concurrent atomic computations associated with a given output or hidden unit. These operations can be broken down into smaller subtasks depending on the degree of parallelism sought. For instance, a virtual output-layer node may represent a weighted-sum of its inputs $(\sum_{i=1}^{N} w_{li} x_i)$. Clearly, this operation can be broken down into multiplications and additions.

Each pipelined training phase can be expressed in terms of multiple passes through the *PCC* network. For instance, each presentation of the training set during the output-unit learning phase consists of $m + H$ passes through the *PCC* network. Based on the *PCC* model, output-unit training proceeds as follows. During the k^{th} pass, virtual output-layer units receive outputs of the virtual input-layer units and compute the products corresponding to the k^{th} pattern. Then, every output unit (of the actual *CC* network) computes its output for pattern k-H based on the products computed in previous iterations. At the same time, each hidden unit, say unit r ($1 \le r \le H$), computes its output for pattern

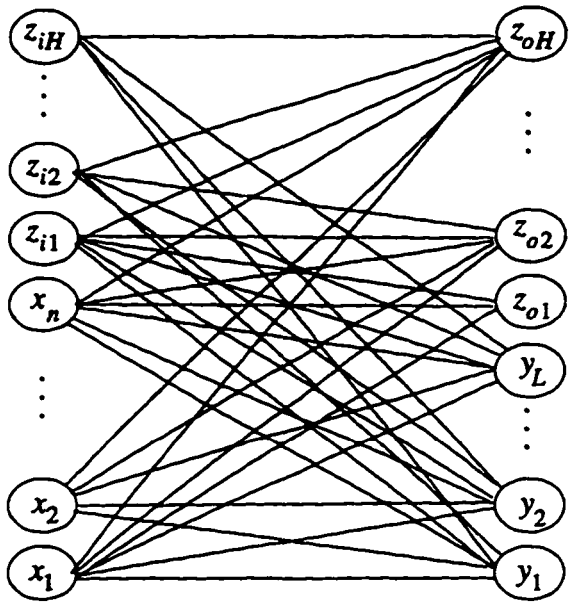

Figure 3.2: The *PCC* model.

k-r. Finally, hidden unit z_{or} (in the virtual output layer) sends its output for pattern *k-r* to z_{ir} (in the virtual input layer) for subsequent operations. Clearly, outputs of each pattern are computed after *H* passes through the *PCC* network. Notice that multiple patterns (at most *H* patterns) are processed concurrently in the pipelined algorithm.

Due to the nature of the pipelined algorithm, computations and communications in multiple passes of the *PCC* network cannot be overlapped. Hence, the problem of mapping the pipelined algorithm on the *KNC* is simplified to that of mapping the *PCC* model, which is described next.

3.2.2 The Mapping Procedure

In this subsection we consider mapping the *PCC* for a network with *Hmax* hidden units to an n_a-dimensional *KNC*. Our objective is to find a mapping scheme which leads to an efficient simulation of the *CC* learning algorithm. In this algorithm, the number of hidden

units is not known in advance. Hidden units are installed one by one until a desired error bound is reached. Consequently, the size and connectivity of the *PCC* change dynamically. One possible approach is to devise an efficient simulation for each instance of the *PCC*. However, such a scheme would require task reassignment whenever a new hidden unit is added. This would lead to a very inefficient implementation due to the time wasted in reassignment. Here, we propose an efficient method which eliminates the need for task reassignment and minimizes a desired metric for a network with *Hmax* hidden units. Instead of mapping all instances of the *PCC*, we consider mapping a single *PCC* which captures (as shall be seen) the features of all possible instances of the *PCC* (within certain limits).

We consider an upper bound on the number of hidden units (H) and denote it by *Hmax*. We assume that H can take any value between 0 and H_{max} with equal probability. Then, we map the *PCC* with *Hmax* hidden units (denoted by PCC_{max} henceforth) such that a desired metric is minimized. In particular, we propose two optimization metrics for a network with *Hmax* hidden units and compare the performance of their resulting mappings. In our approach, the processor assignment for *PCC*'s with 0 through *Hmax* hidden units is known in advance. In other words, the tasks each processor should perform for any of these network sizes are known once the mapping is done. Moreover, no task rescheduling or migration is needed as the number of hidden units grows. Each processor basically needs to compute more tasks as new hidden units are allocated. The mapping of the PCC_{max} takes place in two steps. We first map the PCC_{max} to a virtual *KNC* of compatible size, called *VKNC*. Then, we fold the *VKNC* until its dimension matches that of the actual *KNC*. The

folding is done in such a way so as to optimize a desired metric for a network with *Hmax* hidden units. Further foldings maybe necessary if a subcube of the parallel architecture minimizes the desired metric. In such cases the foldings continue until the metric is minimized.

3.2.3 Mapping the PCC_{max} onto a *VKNC*

The *PCC* learning model involves both communications as well as computations. The mapping should satisfy both communication and computation requirements in such a way that the a desired metric is minimized for a network with *Hmax* hidden units. We need to find the degree of parallelism which leads to minimization of the metric assuming *Hmax* hidden units. The granularity of the parallelism should be chosen based on two factors: the ratio of communication time to computation time of the actual *KNC* (t_c / t_r) and its dimensionality (n_a).

Our approach is to first obtain the task assignment for the finest grain parallelism; i.e., each simple arithmetic operation is assigned to a virtual processor. Each PCC_{max} node represents a computation which can be partitioned into atomic subtasks. Notice that the number of atomic computations associated with each node is related to the number of edges connected to the node. Such tasks are assigned to processors of the virtual *KNC*. For instance, each multiplication in $\sum_{i=1}^{N} w_{li} x_i$ is considered an atomic task and is assigned to a *VKNC* node. Then, we refine the granularity of parallelism by folding the virtual architecture thus increasing the amount of computation performed by each processor at least until the number of concurrent tasks matches the size of the actual *KNC*. Further foldings might be necessary to optimize the chosen metric when *Hmax* hidden units are allocated.

For the finest parallelism, we assign each pair of adjacent units of the PCC_{max}, (which represents an atomic task), to one node of the *VKNC*. Notice that the finest grain parallelism may not lead to minimization of the desired metric. However, we use it as the starting point of our mapping. The degree of granularity is increased as the mapping proceeds. An atomic task in this work represents a simple arithmetic operation such as addition, subtraction, or multiplication. This way, the local memory of the *VKNC* nodes provide the communication between any two adjacent units. Since each PCC_{max} unit has several neighbors, multiple copies of each node would be assigned to different *VKNC* nodes. The *KNC* interconnection network will provide communications among different copies of each unit. The granularity of the parallelism is refined during the folding process.

We assign an n_{max}-digit (l_{max}-digit) k-ary address to each input-layer (output-layer) unit of the PCC_{max}, where $n_{max} = \lceil \log_k (N + Hmax) \rceil$ and $l_{max} = \lceil \log_k (L + Hmax) \rceil$. These labels are used to uniquely identify the atomic computations associated with each pair of adjacent PCC_{max} units. PCC_{max} units are then assigned to an $(n_{max} + l_{max})$-dimensional *VKNC* as follows. A copy of adjacent PCC_{max} units i and j are assigned to the *VKNC* node whose address is obtained by concatenating the addresses of units i and j. The *VKNC* node performs all computations involving both of these units. The assignment implies storing in the *VKNC* node (or in some cases initializing) several parameters associated with units i and j. We refer to these parameters as the *data sets* of *VKNC* nodes. Table 3.2 shows possible data sets for different adjacent nodes of PCC_{max}. Each column in this table represents a different data set.

Table 3.2: Data sets for different adjacent units of PCC_{max}

Input x_i and output y_j	Input x_i and hidden z_j	Hidden units z_i and z_j	Hidden z_i and output y_j
$x_i^1, x_i^2, ..., x_i^k$	$x_i^1, x_i^2, ..., x_i^k$	v_{ji}	$d_j^1, d_j^2, ..., d_j^k$
$y_j^1, y_j^2, ..., y_j^k$	p_{ji}	z_j	$y_j^1, y_j^2, ..., y_j^k$
$d_j^1, d_j^2, ..., d_j^k$		z_i	q_{ji}
w_{ji}			

This assignment process is repeated for each pair of adjacent PCC_{max} units. It is easy to show that with this assignment scheme all copies of any virtual output-layer unit will lie on an n_{max}-dimensional *KNC*. In all, there are l_{max} edge-disjoint n_{max}-dimensional *KNC*'s which will provide the communications corresponding to all virtual output units. We refer to these *KNC's* as *input-to-output* subcubes. Notice that in each phase of the algorithm certain sum-of-products operations are performed which correspond to the computations that would be performed by each hidden or output unit. We utilize input-to-output subcubes to provide the communications necessary for such operations. Similarly, *output-to-input* subcubes, provide communications to the virtual input units.

3.2.4 Folding the *VKNC*

The next step in the mapping process involves reducing the size of the *VKNC* until it is less than or equal to that of the actual architecture, depending on the ratio of communication time to computation time. We refer to this process as *folding* the *VKNC*. Our objective is to fold the *VKNC* of a network with *Hmax* hidden units such that the resulting mapping leads to minimization of a desired metric for the given network. We

consider two optimization metrics defined based on the *iteration time* of the learning algorithm for a network with *Hmax* hidden units, which is defined as the total execution time taken by one iteration of the two learning phases. Let n_a and n_v denote the sizes of the actual *KNC* and the *VKNC*, respectively. We refer to the architecture obtained after folding the *VKNC* t times by $VKNC^t$. In the proposed scheme the *VKNC* undergoes at least n_f folding steps where $n_f = n_v - n_a$. At folding step $t+1$, the dimensionality of $VKNC^t$ is reduced by one, and tasks and data sets of all nodes whose k-ary addresses differ only in the digit corresponding to the folding dimension are assigned to a single node of the $VKNC^{t+1}$. Any desired communication among these tasks would take place through the local memory of the assigned node. The issue here is how to select the folding digits (dimensions) to satisfy the optimization criterion.

The folding digit can be selected from the input segment digits or the output segment digits of the $(n_{max} + l_{max})$-digit *VKNC* address. In Chapter 2 we have shown that folding the most significant digit from each segment of the *VKNC* label maximizes processor utilization. We need to determine how many digits should be folded from each segment. It should be pointed out that we fold the *VKNC* obtained for a network with *Hmax* hidden units. However, for subsequent derivations we need to determine how such a given folding step affects smaller instances of the network. Figure 3.3 shows all possible ways folding the *VKNC* can affect different instances of the model. Let $l_h = \lceil \log_k (L + H) \rceil$ and $n_h = \lceil \log_k (N + H) \rceil$. In general, folding α input digits of the *VKNC* address of a network with *Hmax* hidden units reduces the size of the input segment of other PCC_H's $(0 \le H \le Hmax-1)$ by $\max(\alpha - n_{max} - n_h, 0)$ digits. Similarly, folding β output digits from

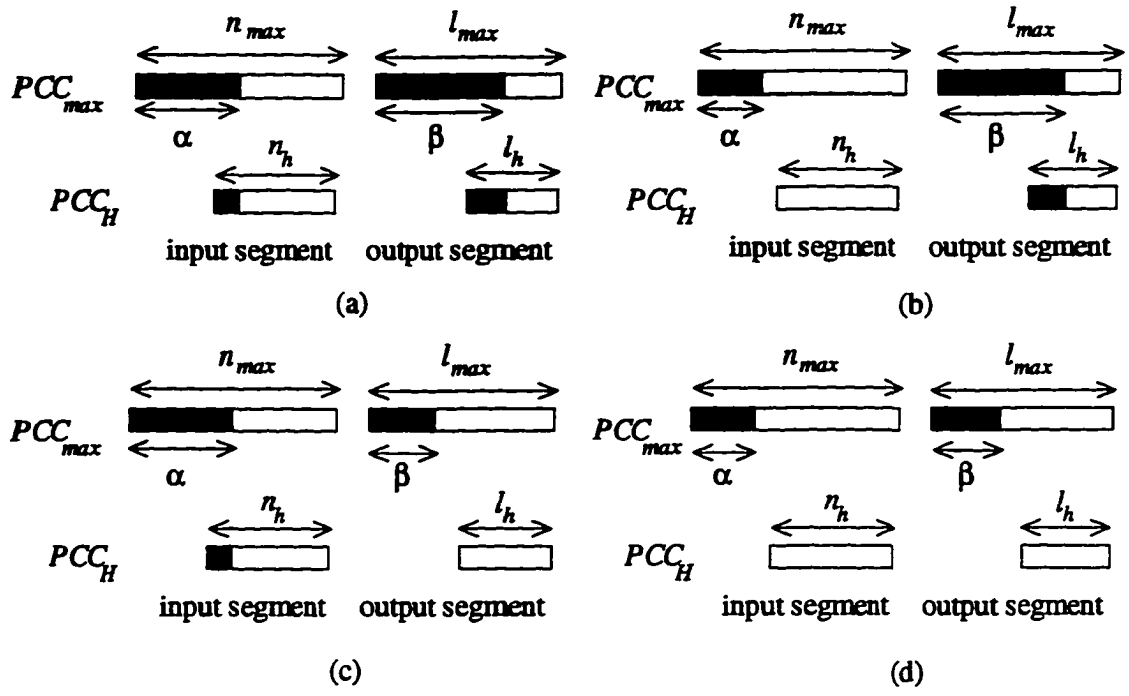

Figure 3.3: Input and output segments of PCC_{max} and PCC_H after α input and β output digit foldings.

the *VKNC* address of a network with *Hmax* hidden units reduces the size of the output segment of other PCC_H's by $\max(\beta - l_{max} - l_h, 0)$ times. Next, we find the iteration time for any instance of a network (with 0 through *Hmax* hidden units) if the *VKNC* of the largest instance of the network (with *Hmax* hidden units) undergoes certain number of foldings. We wish to stress on the fact that the folding scheme reduces the size of the *VKNC* of a network with *Hmax* hidden units.

Theorem 3.1: If the *CC* output-unit training algorithm is to be run on a *KNC* whose size equals that of $VKNC^{\alpha+\beta}$, obtained after α input-segment and β output-segment foldings of the *VKNC*'s label, then with m input patterns each iteration of a network instance with H hidden units will take

$$T_H^{output}(\alpha, \beta) = (m+H)\max(k^{\beta - l_{max} + l_h}, 1)\Big[(1 + 8\max(k^{\alpha - n_{max} + n_h}, 1)) t_r + \min(n_h, n_{max} - \alpha)(\lceil k/2 \rceil t_c + 2t_r) + \big(\min(n_h, n_{max} - \alpha) + c_1 \min(l_h, l_{max} - \beta)\big)\lceil k/2 \rceil t_c \Big] \quad (3.2)$$

time units, where $c_1 = \dfrac{\max(k^{\alpha - n_{max} + n_h}, 1)}{\max(k^{\beta - l_{max} + l_h}, 1)}$.

Proof: This phase consists of two major steps (Steps 1 and 2 of Algorithm 3.1). Step 1 consists of Steps 1.1 and 1.2 which are executed in parallel and involve a sum-of-products computation. It has been shown in Chapter 2 that the minimum time to compute the sum of products of k^n pairs on an i-dimensional *KNC* ($i \le n$) is

$$T(i) = i(\lceil k/2 \rceil t_c + 2 t_r) + (2 k^{n-i} - 1) t_r \quad (3.3)$$

A *fan-in* algorithm is introduced in Chapter 2 (Algorithm 2.2) which achieves this lower bound. The dimension of the largest subcube of the folded *VKNC* involved in Step 1.1.2 and Step 1.2.2 is $\min(n_h, n_{max} - \alpha)$ (see Figure 3.3). The output segment of the *VKNC* label is folded $\max(\beta - l_{max} + l_h, 0)$ times. So, each of the $\min(n_h, n_{max} - \alpha)$-dimensional subcubes has to compute $\max(k^{\beta - l_{max} + l_h}, 1)$ sum of products. Hence, according to (3.3) Steps 1.1.2 and 1.2.2 take at most

$$\max(k^{\beta - l_{max} + l_h}, 1)\left[2\max(k^{\alpha - n_{max} + n_h}, 1)t_r + \min(n_h, n_{max} - \alpha)(\lceil k/2 \rceil t_c + 2t_r)\right] \quad (3.4)$$

time units.

The computations in Steps 1.2.3 and 1.1.3 take $\max(k^{\beta - l_{max} + l_h}, 1)\, t_r$ time units assuming that $G(.)$ and $F(.)$ take t_r time units. The outputs of output units computed in Step 1.2.3 should be broadcasted to other nodes of the corresponding input-to-output subcubes for subsequent operations. The hidden-unit outputs (z_j's) in Step 1.1.3 should be sent to

their corresponding virtual input-layer nodes, $z_{\bar{q}}$'s. According to [10], one-to-all broadcast on a k-ary n-cube can be performed optimally in $n\,k/2\;t_c$ time units when k is even. For the case of odd k it is shown that the one-to-all broadcast can be performed in $n\lceil k/2 \rceil$ time units [10]. Hence, Steps 1.1.3 and 1.2.3 take at most

$$\max(k^{\beta - l_{max} + l_h}, 1)\Big[t_r + \Big(c_1 \min(l_h , l_{max} - \beta) + \min(n_h , n_{max} - \alpha) \Big) \lceil k/2 \rceil t_c \Big] \qquad (3.5)$$

time units.

Each weight increment in Step 1.2.4 involves 6 simple mathematical operations, requiring $6t_r$ time units. This step is repeated $\max(k^{\beta - l_{max} + l_h}, 1).\max(k^{\alpha - n_{max} + n_h}, 1)$ times due to the folding. Hence, Step 1.2.4 takes

$$6\max(k^{\beta - l_{max} + l_h}, 1).\max(k^{\alpha - n_{max} + n_h}, 1)t_r \qquad (3.6)$$

time units.

Step 1 is repeated $m+H$ times during each iteration due to pipelining. Therefore, using (3.4) through (3.6) we conclude that Step 1 takes

$$\begin{aligned} (m+H)\ & \max(k^{\beta - l_{max} + l_h}, 1) \\ & \Big[\max\big(\min (n_h , n_{max} - \alpha) , c_1 \min(l_h , l_{max} - \beta)\big) \lceil k/2 \rceil t_c + \\ & \min (n_h , n_{max} - \alpha)(\lceil k/2 \rceil t_c + 2t_r) + \\ & (1 + 8\max(k^{\alpha - n_{max} + n_h}, 1))\, t_r \Big] \end{aligned} \qquad (3.7)$$

time units.

Step 2.1 takes $m.\max(k^{\beta - l_{max} + l}, 1)t_r$ time units. To compute $\overline{E_l}$ in Step 2.2, first residual errors are added locally in each node. This takes $m.\max(k^{\beta - l_{max} + l}, 1)t_r$ time units. Then, using output-to-input subcubes of dimension $\min(l , l_{max} - \beta)$, $\overline{E_l}$ is calculated and

broadcasted to each output node in $2\min(l, l_{max} - \beta)(\lceil k/2 \rceil t_c + t_r)$ time units. Hence, Step 2 takes

$$2\Big(m.\min(k^{\beta - l_{max} + l}, 1)t_r + \min(l, l_{max} - \beta)(\lceil k/2 \rceil t_c + t_r)\Big) \tag{3.8}$$

time units.

Step 1 is repeated a certain number of iterations to see if a desired error bound can be reached. Step 2 on the other hand is performed only once to compute the residual error when the desired error bound is not reached in Step 1. Moreover, by comparing (3.7) and (3.8) we observe that the time taken by Step 2 is negligible. Hence, we approximate the iteration time of the output-unit training by the time taken by Step 1. □

Theorem 3.2: If the *CC* hidden-unit training algorithm is to run on a *KNC* whose size equals that of $VKNC^{\alpha+\beta}$, obtained after α input-segment and β output-segment foldings of the *VKNC*'s label, then for a network instance with H hidden units and m input patterns each iteration will take

$$\begin{aligned} T_H^{hidden}(\alpha,\beta) = (m + H) \max(k^{\beta - l_{max} - l_h}, 1) \\ \Big[\Big(2\max(k^{\alpha - n_{max} + n_h}, 1) + 3c_2 + c_1 c_2 \Big) t_r + \\ \min(l_h, l_{max} - \beta)\Big((c_1 + c_2)\lceil k/2 \rceil t_c + c_2 t_r\Big) + \\ \min(n_h, n_{max} - \alpha)\Big((c_2 + 1)\lceil k/2 \rceil t_c + 2t_r\Big)\Big] \end{aligned} \tag{3.9}$$

time units, where $c_1 = \dfrac{\max(k^{\alpha - n_{max} + n_h}, 1)}{\max(k^{\beta - l_{max} + l_h}, 1)}$ and $c_2 = \dfrac{m}{m + H}$.

Proof: Step 1 of hidden unit training is similar to Step 1.1 of the output-unit training phase (see Algorithm 3.2). This step is repeated $m + H$ times. Hence, Step 1 takes

$$\begin{aligned} (m + H) \max(k^{\beta - l_{max} + l_h}, 1)\Big[2\max(k^{\alpha - n_{max} + n_h}, 1)\, t_r + \\ c_1 \min(l_h, l_{max} - \beta)\lceil k/2 \rceil t_c + \min(n_h, n_{max} - \alpha)\,(\lceil k/2 \rceil t_c + 2t_r)\Big] \end{aligned} \tag{3.10}$$

Step 2 is repeated m times. The product $\sigma_l(E_l^k - \overline{E_l})$ in Steps 2.1 and 2.2 for each node can be computed and added locally in $3\max(k^{\beta - l_{max} - l_h}, 1)t_r$ time units. The overall sum of these local values for each pattern can be computed using the output-to-input subcubes of dimension $\min(l, l_{max} - \beta)$. Hence, Steps 2.1 and 2.2 take

$$\max(k^{\beta - l_{max} - l_h}, 1)\left[3t_r + \min(l, l_{max} - \beta)(\lceil k/2 \rceil t_c + t_r)\right] \tag{3.11}$$

time units. The sums should be broadcast to each node of the input-to-output subcubes associated with the new hidden units. Finally, $\rho \sum \sigma_l(E_l^{k-r} - \overline{E_l})$ is multiplied by $(\partial G(net_r^{k-r})/\partial x_i^{k-r})\, x_i^{k-r}$ in t_r time units. Hence, Step 2 takes:

$$\begin{aligned} &m.\max(k^{\beta - l_{max} - l_h}, 1)\left[(3 + c_1)t_r + \min(l, l_{max} - \beta)(\lceil k/2 \rceil t_c + t_r)\right] + \\ &\min(n_h, n_{max} - \alpha)\lceil k/2 \rceil t_c \end{aligned} \tag{3.12}$$

time units. Using (3.10) and (3.12) we can express the overall iteration time for hidden unit training as in (3.9). □

Let $T_H(\alpha, \beta)$ denote the iteration time of the pipelined training algorithm for an instance of the network with H hidden units if α and β digits have been folded from the input and output segments of the *VKNC* label of the largest network, respectively. By definition, $T_H(\alpha, \beta)$ is obtained by adding $T_H^{output}(\alpha, \beta)$ and $T_H^{hidden}(\alpha, \beta)$. Hence,

$$\begin{aligned} T_H(\alpha, \beta) = {} & (m + H)\max(k^{\beta - l_{max} + l_h}, 1) \\ & \Big[(10\max(k^{\alpha - n_{max} + n_h}, 1) + 3c_2 + k^{\alpha-\beta}c_2 + 1)t_r + \\ & \min(n_h, n_{max} - \alpha)((c_2 + 3)\lceil k/2 \rceil t_c + 4t_r) + \\ & \min(l_h, l_{max} - \beta)((2k^{\alpha-\beta} + c_2)\lceil k/2 \rceil t_c + c_2 t_r)\Big] \end{aligned} \tag{3.13}$$

We introduce two optimization metrics and show that a mapping based on either of these criteria leads to a very efficient simulation of the learning algorithm, although one metric is superior to the other. Our approach is based on the hypothesis that optimizing the mapping for a network with a known size will lead to near-optimal mappings for other instances of the network. We shall show that our hypothesis is indeed valid.

The first optimization metric is T_{Hmax}, the iteration time of a network with *Hmax* hidden unit. The other criterion is the sum of iteration times of all instance of a network with 0 through *Hmax* hidden units. We denote this metric by $T^{total}_{Hmax}(\alpha,\beta) = \sum_{H=0}^{Hmax} T_H(\alpha,\beta)$. Notice that this metric is closely related to the average iteration time of the network assuming that *H* can take any value from 0 through *Hmax* with equal probability.

Next we present our analytical approaches for mapping a *CC* network on a given *KNC* parallel architecture based on the above criteria. We will analyze the computational complexity of each approach. Then, we demonstrate the efficiency of the mapping based on each of the two metrics.

3.2.4.1 Optimizing $T_{Hmax}(\alpha, \beta)$ for a Network with *Hmax* Hidden Units

We need to determine the iteration time of the largest instance of the network. By substituting *Hmax* in (3.13) we obtain

$$\begin{aligned} T_{Hmax}(\alpha,\beta) = \; & (m + Hmax)\, k^{\beta} \\ & \Big[(10k^{\alpha} + 3c_2 + c_2 k^{\alpha-\beta} + 1)t_r + \\ & (n_{max}-\alpha)\,((c_2 + 3)\lceil k/2 \rceil t_c + 4t_r) + \\ & (l_{max}-\beta)\,((2k^{\alpha-\beta} + c_2)\lceil k/2 \rceil t_c + c_2 t_r)\Big] \end{aligned} \tag{3.14}$$

where $c_2 = \dfrac{m}{m + Hmax}$. We need to optimize this equation in terms of selection of α and β.

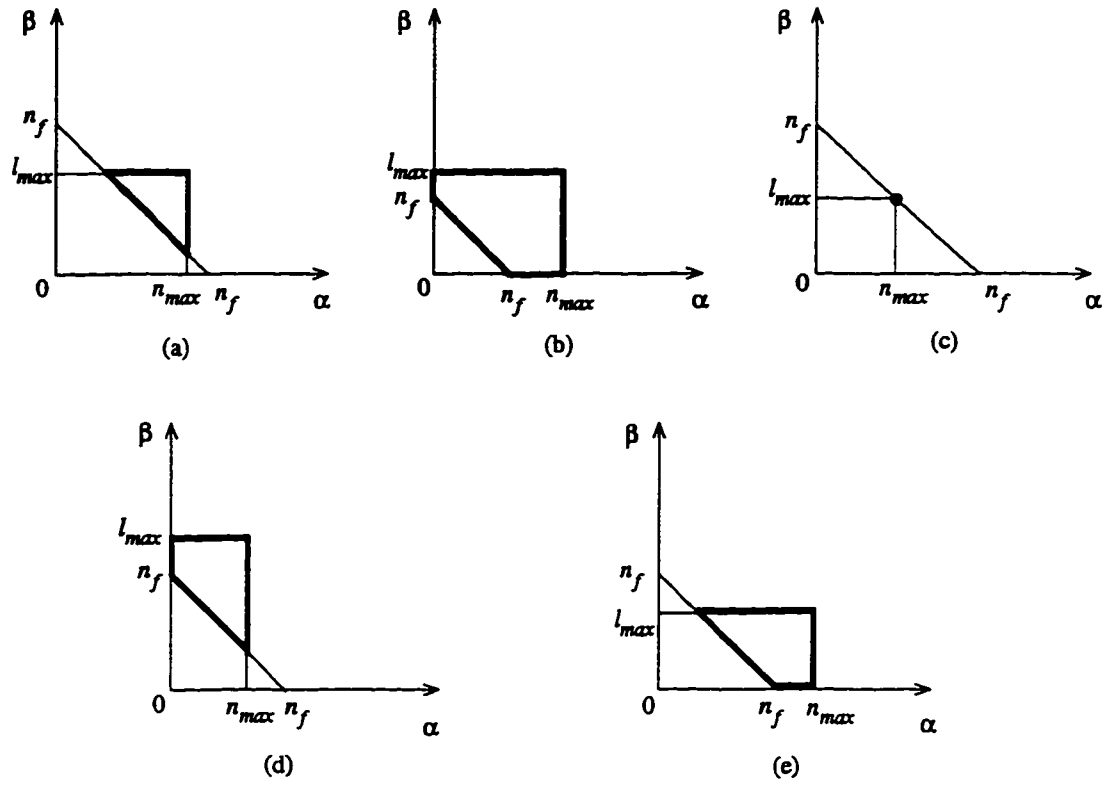

Figure 3.4: Possible closed sets for the optimization problem.

We know that $\alpha + \beta \geq n_f$, where $n_f = n_v - n_a$. Additional foldings might be necessary depending on the ratio of t_c/t_r. This optimization problem is basically a search for α and β in a closed set interval; specifically one of the five possible sets shown in Figure 3.4. This is essentially a nonlinear integer programming problem. Because of the nature of the function it is difficult to use typical optimization techniques. We introduce a computationally efficient method for finding α and β that yields optimal mapping by utilizing properties of the function. Notice that by considering all points in the search space (shown in Figure 3.4), the size of the folded graph will be less than or equal to n_a, depending on which size yields minimum value of T_{Hmax}. Hence, the folding approach may only select a subcube of the parallel architecture. Since we want to minimize the function in terms of

both α and β, we can first minimize it in terms of α for different β value. Then, we can minimize the function in terms of β using the α values obtained for each β value. This is the approach we adopt here.

Theorem 3.3: $T_{Hmax}(\alpha,\beta)$ is a convex function of α.

Proof: The first partial derivative of this function with respect to α are given by

$$\frac{\partial T_{Hmax}(\alpha,\beta)}{\partial \alpha} = (m + Hmax)\, k^{\beta} \left[(10 \ln k\, k^{\alpha} + c_2 \ln k\, k^{\alpha-\beta}) t_r - ((c_2 + 3)\lceil k/2 \rceil t_c + 4t_r) + (l_h - \beta)\, 2 \ln k\, k^{\alpha-\beta} \lceil k/2 \rceil t_c\right] \tag{3.15}$$

The second partial derivative with respect to α is as follows:

$$\frac{\partial^2 T_{Hmax}(\alpha,\beta)}{\partial \alpha^2} = (m + Hmax)\, k^{\beta} \ln^2 k \left[(10\, k^{\alpha} + c_2\, k^{\alpha-\beta}) t_r + (l_{max} - \beta)\, 2\, k^{\alpha-\beta} \lceil k/2 \rceil t_c\right] \tag{3.16}$$

The right hand side of (3.16) is always positive. Hence, the function is convex with respect to α. □

Theorem 3.4: For a given β, the value of α which minimizes T_{Hmax} is given by

$$\alpha = \begin{cases} \lceil Q \rceil & \text{if } \max(0, n_f - l_{max}) < Q < n_{max} \text{ and } T_{H_{max}}(\lceil Q \rceil, \beta) < T_{H_{max}}(\lfloor Q \rfloor, \beta) \\ \lfloor Q \rfloor & \text{if } \max(0, n_f - l_{max}) < Q < n_{max} \text{ and } T_{H_{max}}(\lceil Q \rceil, \beta) > T_{H_{max}}(\lfloor Q \rfloor, \beta) \\ \max(0, n_f - l_{max}) & \text{if } Q \le \max(0, n_f - l_{max}) \\ n_{max} & \text{if } n_{max} \le Q \end{cases} \tag{3.17}$$

Parameter Q in (3.17) is obtained as follows:

$$Q = \log_k \frac{(c_2 + 3)\lceil k/2 \rceil t_c + 4 t_r}{\ln k \left[(10 + c_2 k^{-\beta}) t_r + 2 (l_{max} - \beta) k^{-\beta} \lceil k/2 \rceil t_c \right]} \tag{3.18}$$

Proof: $T_{Hmax}(\alpha,\beta)$ is a convex function with respect to α (See Theorem 3.3). From basic calculus we know that for a fixed value of β, the minimum of such a function is either the critical point or one of the two end points of the α interval. Figure 3.3 shows the range of α to be $\alpha \in [\max(0, n_f - l_{max}), n_{max}]$. The critical point of the function is obtained by setting $\partial T_{H_{max}} / \partial \alpha = 0$. Thus the critical point is at

$$\alpha = \log_k \frac{(c_2 + 3)\lceil k/2 \rceil t_c + 4 t_r}{\ln k \left[(10 + c_2 k^{-\beta}) t_r + 2 (l_{max} - \beta) k^{-\beta} \lceil k/2 \rceil t_c \right]} \tag{3.19}$$

We denote the right hand side of (3.19) by Q. Notice that we are looking for an integer value of α. Hence, for a fixed β, the minimum of $T_{Hmax}(\alpha,\beta)$ occurs at

$$\alpha = \begin{cases} \lceil Q \rceil & \text{if } \max(0, n_f - l_{max}) < Q < n_{max} \text{ and } T_{H_{max}}(\lceil Q \rceil, \beta) < T_{H_{max}}(\lfloor Q \rfloor, \beta) \\ \lfloor Q \rfloor & \text{if } \max(0, n_f - l_{max}) < Q < n_{max} \text{ and } T_{H_{max}}(\lceil Q \rceil, \beta) > T_{H_{max}}(\lfloor Q \rfloor, \beta) \\ \max(0, n_f - l_{max}) & \text{if } Q \le \max(0, n_f - l_{max}) \\ n_{max} & \text{if } n_{max} \le Q \end{cases} \tag{3.20}$$

□

Hence, we can find the values of α and β that yield optimal $T_{Hmax}(\alpha,\beta)$ by finding the best value for α for each β value, and then find the best β among those found. As the next theorem shows, this process has logarithmic time complexity.

Theorem 3.5: The search for the minimum value of $T_{Hmax}(\alpha,\beta)$ has computational complexity $O(\log(L + H_{max})) = O(l_{max})$.

Proof: Clearly, we can find the minimum of the function for each β in constant time using (3.20). We can repeat the above process for each β. Notice that $\beta \le l_{max}$. Hence, the overall search has computational complexity $O(l_{max})$. □

3.2.4.2 Optimizing $T^{total}_{Hmax}(\alpha,\beta)$ Assuming a Network with *Hmax* Hidden Units

The optimization criterion stated in the previous subsection takes into account only the iteration time of the largest possible network. The resulting mapping might not be efficient for smaller networks. Notice that the number of hidden units is not known in advance. Here we consider an optimization metric defined based on the sum of iteration times of all instances of a network with 0 through *Hmax* hidden units. We denote this metric by $T^{total}_{Hmax}(\alpha,\beta)$ where $T^{total}_{Hmax}(\alpha,\beta) = \sum_{H=0}^{Hmax} T_H(\alpha,\beta)$. Now we introduce an analytical approach for minimizing $T^{total}_{Hmax}(\alpha,\beta)$. As before, We can find the minimum by first minimizing the function with respect to α. Then, we find the β which results in the overall minimum.

We define four sets P_i, $1 \le i \le 4$, as follows:

$$\begin{aligned} P_1 &= \{H \mid 0 \le H \le Hmax,\ l_h \le l_{max} - \beta,\ \text{and } n_h \le n_{max} - \alpha\} \\ P_2 &= \{H \mid 0 \le H \le Hmax,\ l_h \le l_{max} - \beta,\ \text{and } n_h > n_{max} - \alpha\} \\ P_3 &= \{H \mid 0 \le H \le Hmax,\ l_h > l_{max} - \beta,\ \text{and } n_h > n_{max} - \alpha\} \\ P_4 &= \{H \mid 0 \le H \le Hmax,\ l_h > l_{max} - \beta,\ \text{and } n_h \le n_{max} - \alpha\} \end{aligned} \tag{3.21}$$

Theorem 3.6: $T^{total}_{Hmax}(\alpha,\beta)$ is a convex function with respect to α.

Proof: It is easy to show that sets P_1 through P_4 are mutually disjoint and that $P_1 \cup P_2 \cup P_3 \cup P_4 = \{0,1,\ldots,H\}$. Hence,

$$T^{total}_{Hmax} = \sum_{H=1}^{Hmax} T_H = \sum_{H\in P_1} T_H + \sum_{H\in P_2} T_H + \sum_{H\in P_3} T_H + \sum_{H\in P_4} T_H \tag{3.22}$$

The four summation terms in (3.22) can be obtained as follows:

$$\sum_{H\in P_1} T_H(\alpha,\beta) = \sum_{H\in P_1} \Big((m + H_{max}) \Big[(10 + 4c_2 + 1)t_r + n_h ((c_2 + 3)\lceil k/2 \rceil t_c + 4t_r) + l_h ((2 + c_2)\lceil k/2 \rceil t_c + c_2 t_r) \Big] \Big) \tag{3.23}$$

$$\sum_{H\in P_2} T_H(\alpha,\beta) = \sum_{H\in P_2} \Big((m + H_{max}) \Big[(3c_2 + (10 + c_2) k^{\alpha} + 1)t_r + (n_{max} - \alpha) ((c_2 + 3)\lceil k/2 \rceil t_c + 4t_r) + (l_h) ((2 k^{\alpha} + c_2)\lceil k/2 \rceil t_c + c_2 t_r) \Big] \Big) \tag{3.24}$$

$$\sum_{H\in P_3} T_H(\alpha,\beta) = \sum_{H\in P_3} \Big((m + H_{max}) k^{\beta} \Big[(10 (k^{\alpha} + 3c_2 + c_2 k^{\alpha-\beta} + 1)t_r + (n_{max} - \alpha) ((c_2 + 3)\lceil k/2 \rceil t_c + 4t_r) + (l_{max} - \beta) ((2 k^{\alpha-\beta} + c_2)\lceil k/2 \rceil t_c + c_2 t_r) \Big] \Big) \tag{3.25}$$

$$\sum_{H\in P_4} T_H(\alpha,\beta) = \sum_{H\in P_4} \Big((m + H_{max}) k^{\beta} \Big[(10 + 3c_2 + c_2 k^{-\beta} + 1)t_r + n_h ((c_2 + 3)\lceil k/2 \rceil t_c + 4t_r) + (l_{max} - \beta) ((2 k^{-\beta} + c_2)\lceil k/2 \rceil t_c + c_2 t_r) \Big] \Big) \tag{3.26}$$

Notice that $T_H(\alpha,\beta)$ is a constant function of α when H is in sets P_1 or P_2 . Hence, the second partial derivative of $T^{total}_{Hmax}(\alpha,\beta)$ with respect to α is given by equation (3.27).

$$\frac{\partial^2 T_{Hmax}^{total}}{\partial \alpha^2} = \sum_{H \in P_2} \frac{\partial^2 T_H}{\partial \alpha^2} + \sum_{H \in P_3} \frac{\partial^2 T_H}{\partial \alpha^2}$$

$$= \sum_{H \in P_2} \ln^2 k \, k^{\alpha} \left(10 + c_2 t_r + 2 l_h \lceil k/2 \rceil t_c \right) + \tag{3.27}$$

$$\sum_{H \in P_2} \ln^2 k \, k^{\alpha} \left[(10 k^{\beta} + c_2) t_r + 2 (l_{max} - \beta) \right]$$

The right hand side of (3.27) is always positive. Hence, $T_{Hmax}^{total}(\alpha,\beta)$ is a convex function with respect to α.

Theorem 3.7: The minimum value of $T_{Hmax}^{total}(\alpha,\beta)$ for a fixed β occurs at

$$\alpha = \begin{cases} \lceil A \rceil & \text{if } \max(0, n_f - l_{max}) < A < n_{max} \text{ and } T_{Hmax}(\lceil A \rceil, \beta) < T_{Hmax}(\lfloor A \rfloor, \beta) \\ \lfloor A \rfloor & \text{if } \max(0, n_f - l_{max}) < A < n_{max} \text{ and } T_{Hmax}(\lceil A \rceil, \beta) > T_{Hmax}(\lfloor A \rfloor, \beta) \\ \max(0, n_f - l_{max}) & \text{if } A \le \max(0, n_f - l_{max}) \\ n_{max} & \text{if } n_{max} \le A \end{cases} \tag{3.28}$$

where

$$A = \log_k \frac{\ln k^{-1} \, |P_2| \, |P_3| (c_2 + 3) \lceil k/2 \rceil t_c + 4 t_1}{\sum_{H \in P_2} (10 + c_2) t_r + 2 l_h \lceil k/2 \rceil t_c + k^{-\beta} \sum_{H \in P_3} (10 + c_2) t_r + 2 \ln k \, (l_{max} - \beta) \lceil k/2 \rceil t_c} \tag{3.29}$$

Proof: $T_{Hmax}^{total}(\alpha,\beta)$ is a convex function with respect to α (see Theorem 3.6). Hence, we can find the minimum of the function, for each β, from the critical points or the end points of the range of α. The critical points of the function are obtained from $\partial T_{Hmax}^{total}(\alpha,\beta)/\partial\alpha = 0$. This results in

$$\alpha = \log_k \frac{\ln k^{-1} \, |P_2| \, |P_3| (c_2 + 3) \lceil k/2 \rceil t_c + 4 t_1}{\sum_{H \in P_2} (10 + c_2) t_r + 2 l_h \lceil k/2 \rceil t_c + k^{-\beta} \sum_{H \in P_3} (10 + c_2) t_r + 2 \ln k \, (l_{max} - \beta) \lceil k/2 \rceil t_c} \tag{3.30}$$

We denote the right hand side of (3.30) by A. Since we are looking for an integer α, we need to use $\lceil A \rceil$ or $\lfloor A \rfloor$, whichever results in a lower value when substituted in $T^{total}_{Hmax}(\alpha,\beta)$. Hence, we find the value for α which minimizes the function for each β as follows:

$$\alpha = \begin{cases} \lceil A \rceil & \text{if } \max(0, n_f - l_{max}) < A < n_{max} \text{ and } T_{Hmax}(\lceil A \rceil, \beta) < T_{Hmax}(\lfloor A \rfloor, \beta) \\ \lfloor A \rfloor & \text{if } \max(0, n_f - l_{max}) < A < n_{max} \text{ and } T_{Hmax}(\lceil A \rceil, \beta) > T_{Hmax}(\lfloor A \rfloor, \beta) \\ \max(0, n_f - l_{max}) & \text{if } A \le \max(0, n_f - l_{max}) \\ n_{max} & \text{if } n_{max} \le A \end{cases} \tag{3.31}$$

□

Clearly, we can find the minimum of the function for each β in constant time. We can repeat the above process for each β. Hence, the computational complexity of this approach is also $O(\log(L + Hmax)) = O(l_{max})$.

3.2.4.3 Performance of the Proposed Folding Scheme

We now evaluate the performance of the proposed mapping. We begin by examining the efficiency of the resulting learning simulation. We map the learning algorithm for a network with *Hmax* hidden units using the proposed folding scheme. Then, we apply the resulting mapping to all instances of the network (networks with 0 through *Hmax* hidden units) and compute their iteration times. *Iteration time* is the execution time of one iteration of the learning algorithm. We then compare the iteration time of each instance of the network when mapped using our proposed scheme with its corresponding optimal iteration time. We stress on the fact that our scheme optimizes a desired metric only

for a network with *Hmax* hidden units. The optimal iteration time for other instances of the network is obtained through an exhaustive search for comparison purposes only. As we shall show, our mapping leads to near-optimal results for other instances of the network. The results we report here are based on simulation of a network whose parameters are found by running the benchmark application used in [64]. In this case, there are 196 input units, 26 output units, and 1114 training patterns. The average number of hidden units allocated during actual training for this application is reported to be 27. We assume that computation time (t_r) and communication time (t_c) are 0.01 μs and 0.4 μs, respectively. Unless otherwise stated, we assume that the parallel *KNC* architecture has 64 processors.

For the first set of simulations, we performed the folding so as to minimize $T_{Hmax}(\alpha,\beta)$ assuming a network with *Hmax* hidden units. We then used the resulting α and β to compute the iteration times of all instances of the network, with 0 through *Hmax* hidden units. We compared these results with their corresponding optimal values. Figures 3.5 through 3.8 show the results of our simulations for different *Hmax* values; specifically for $H_{max} = 20, 40, 60$, and 80, respectively. Clearly, the iteration times obtained as a result of our mapping are very close to the optimal results except for the noticeable deviation for very small networks. This is to be expected because the optimization criterion takes into account only the iteration time of the largest network. Hence, the resulting mapping might not be efficient for very small networks.

We observe several abrupt changes in the plots depicted in Figures 3.5 through 3.8. These jumps can be attributed to changes in $T_H(\alpha,\beta)$ as H varies. As H grows, several terms in $T_H(\alpha,\beta)$ may vary (see equation (3.13)). Clearly, H grows linearly. n_h and l_h on the other

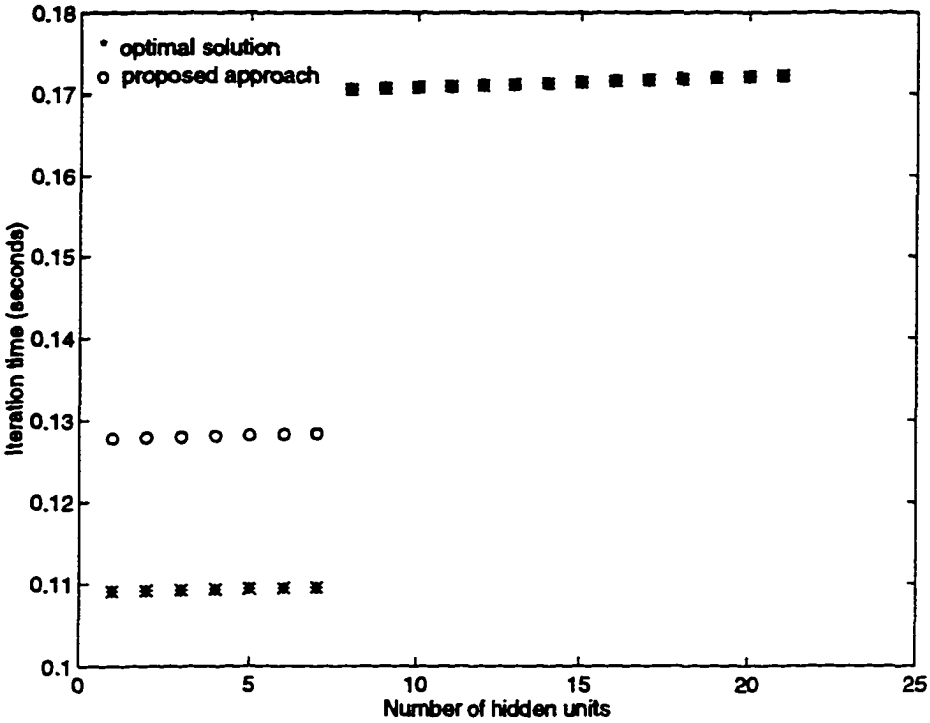

Figure 3.5: Simulation results for optimizing T_{Hmax} with $Hmax = 20$.

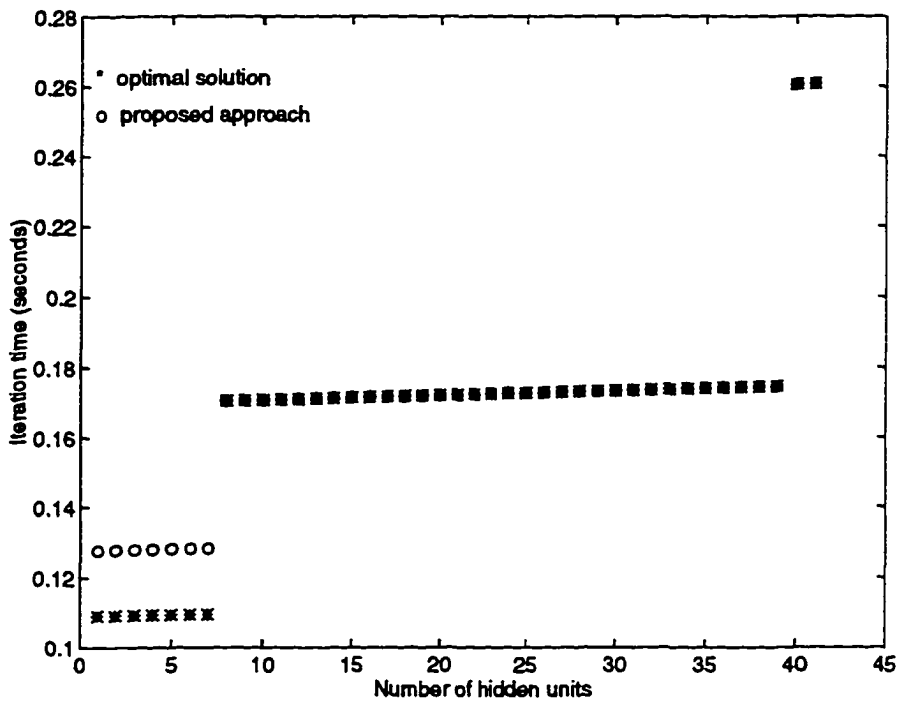

Figure 3.6: Simulation results for optimizing T_{Hmax} with $Hmax = 40$.

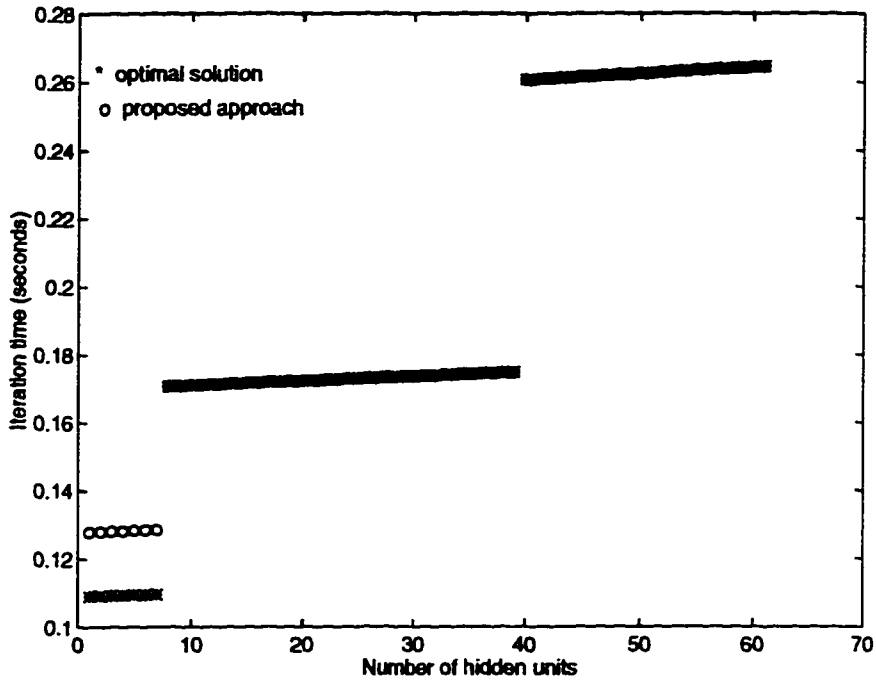

Figure 3.7: Simulation results for optimizing T_{Hmax} with $Hmax = 60$.

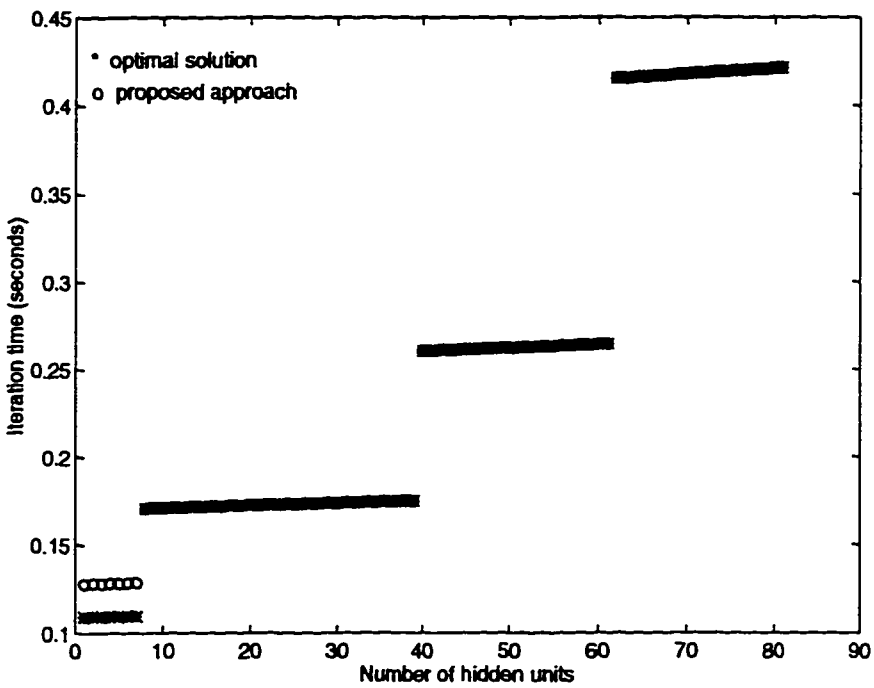

Figure 3.8: Simulation results for optimizing T_{Hmax} with $Hmax = 80$.

hand will change only when $N + H$ or $L + H$ exceed a power of k. For instance, in Figure 3.8 abrupt changes occur at H values 7, 39, and 61. The jumps at H=7 and H=39 are due to an increase in n_h while the jump at H=61 is due to an increase in l_h. Notice that $T_H(\alpha,\beta)$ grows exponentially in terms of n_h and l_h.

Next, we performed the folding by optimizing $T_{Hmax}^{total}(\alpha, \beta)$. We then applied the resulting mapping to all instances of the network with 0 through *Hmax* hidden units and computed $T_H^{total}(\alpha, \beta)$ for each case. We compared these results with their corresponding optimal values obtained by exhaustive search. To obtain the optimal result for a network with H hidden units we performed the mapping to optimize $T_H(\alpha, \beta)$. Figures 3.9 through 3.12 show the results of our simulations with different *Hmax* values. These results indicate that deviation from optimal results appear only for small networks and are relatively insignificant. Comparing these results with those shown in Figures 3.5 through 3.8 we observe that $T_{Hmax}^{total}(\alpha,\beta)$ is a more efficient folding criterion than $T_{Hmax}(\alpha,\beta)$ since it leads to near optimal results for other instances of the network.

For the next set of tests we varied the number of physical processors to examine the speedup of the simulation. Figures 3.13 and 3.14 show how increasing the number of physical processors reduces the iteration time. We observed that speedup eventually saturated. For instance, increasing the number of processors to 8192 did not improve the iteration time when $T_{Hmax}(\alpha,\beta)$ was the optimization metric.

Another issue of relative importance is the selection of *Hmax*. The number of hidden units is determined during the training of the given *CC* architecture. In our mapping scheme we assume an upper bound on the number of hidden units (*Hmax*). Clearly, the efficiency

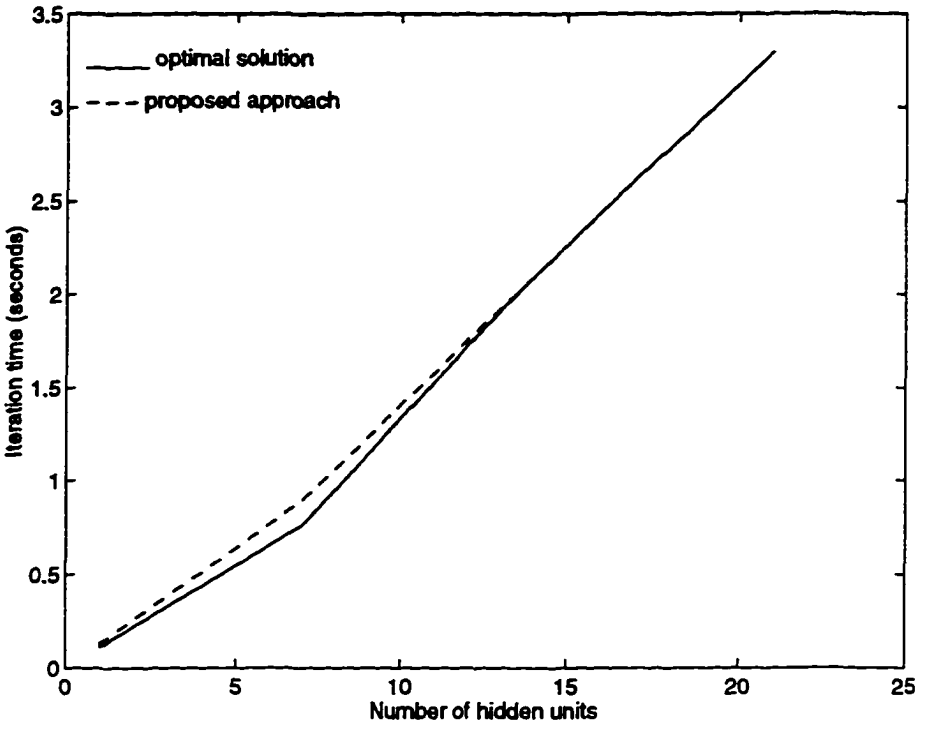

Figure 3.9: Simulation results for optimizing T_{Hmax}^{total} with $Hmax = 20$.

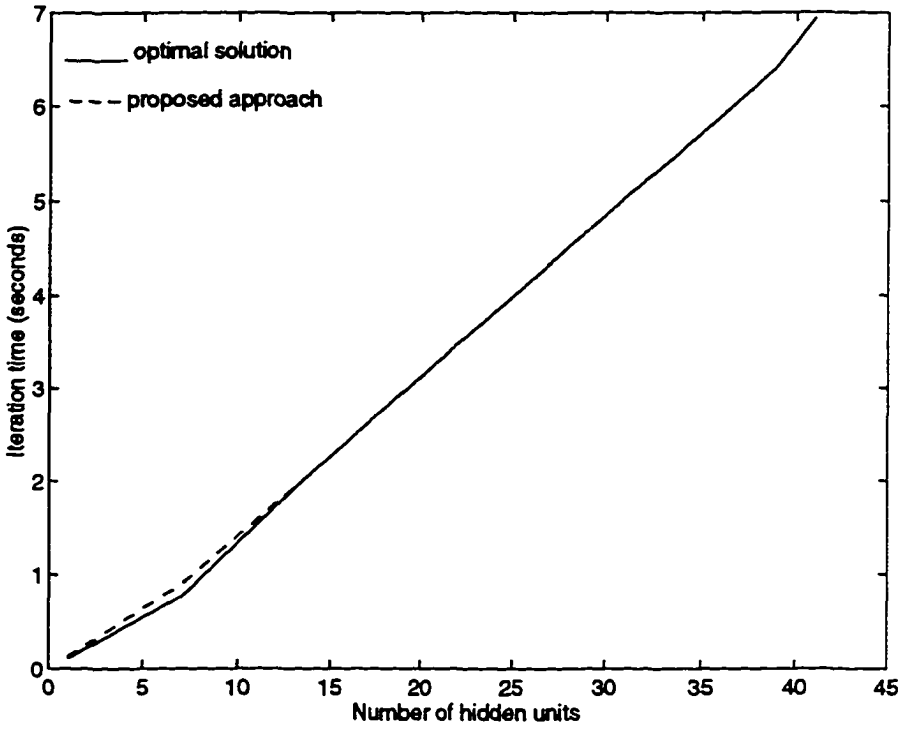

Figure 3.10: Simulation results for optimizing T_{Hmax}^{total} with $Hmax = 40$.

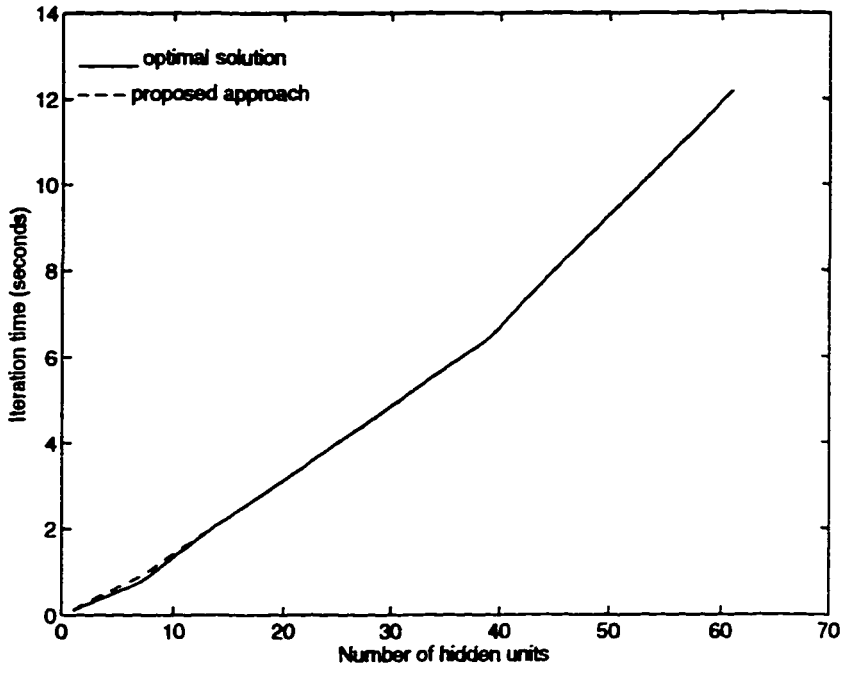

Figure 3.11: Simulation results for optimizing T_{Hmax}^{total} with $Hmax = 60$.

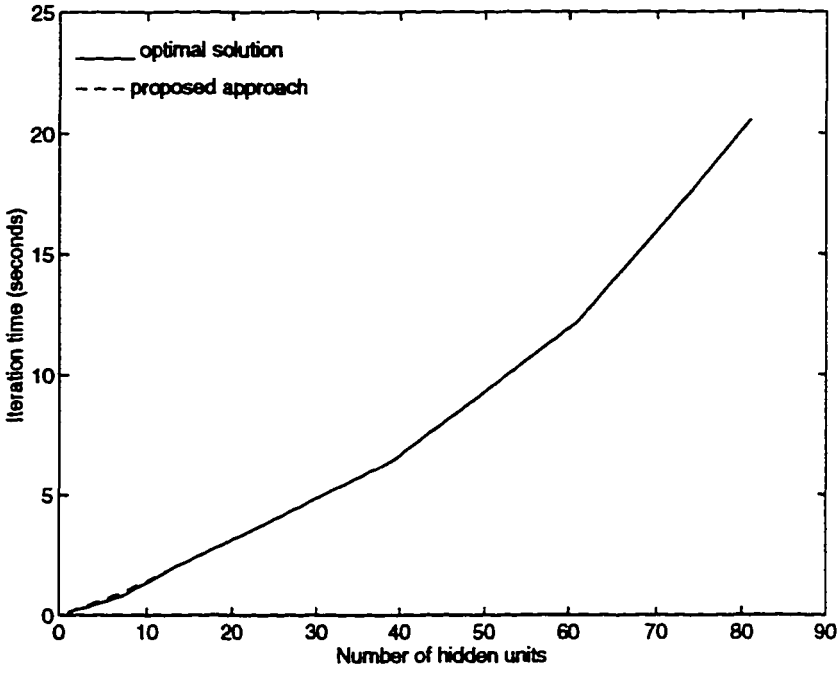

Figure 3.12: Simulation results for optimizing T_{Hmax}^{total} with $Hmax = 80$.

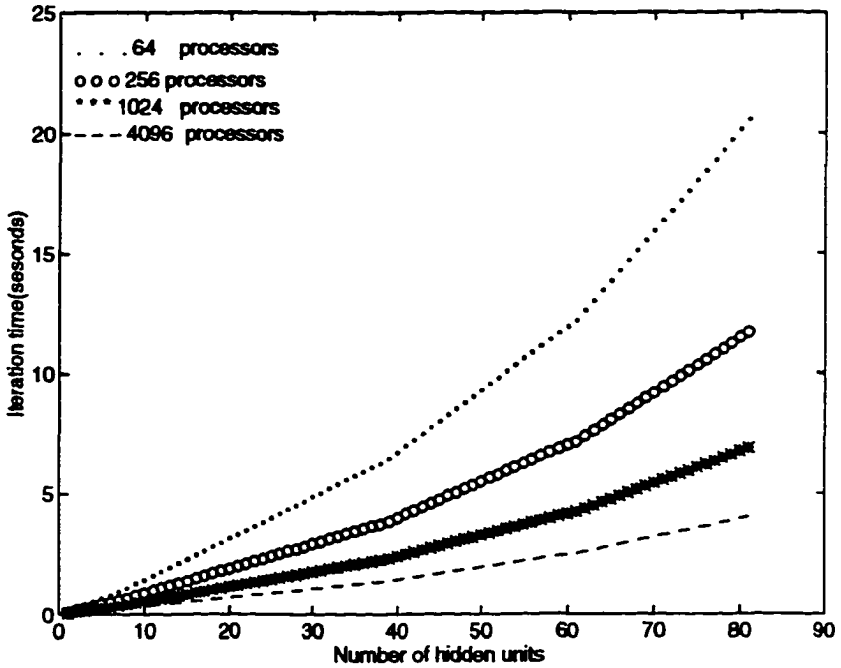

Figure 3.13: The iteration time of *CC* algorithm mapped on *KNC*'s of different sizes when T_{Hmax}^{total} is optimized.

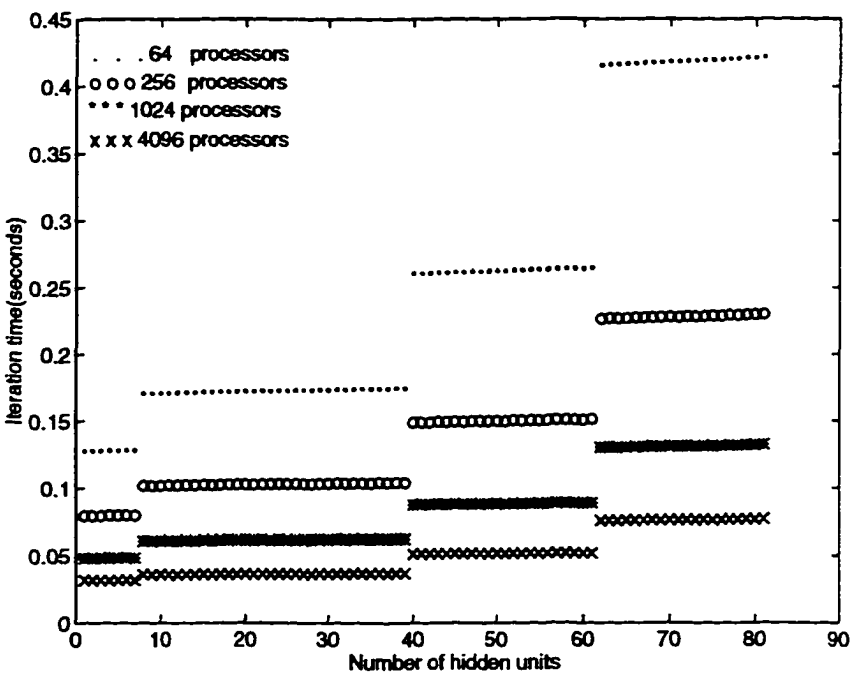

Figure 3.14: The iteration time of *CC* algorithm mapped on *KNC*'s of different sized when T_{Hmax} is optimized.

of the proposed mapping depends on the selection of this value. Next, we explore the effect of *Hmax* on the mapping and on the iteration time of the learning algorithm.

The first issue we considered here was the effect of *Hmax* on performance of smaller instances of the network (with 0 through *Hmax* - 1 hidden units). We mapped a *CC* learning architecture with different *Hmax* values and compared the resulting processor assignments and iteration times. We repeated these simulations for both optimization criteria. The results we report here are based on the NETTalk [64] application mentioned earlier. We assumed a parallel architecture with 32 processors. We used different *Hmax* values, namely 50, 100, and 200. Figure 3.15 shows the iteration times of the sample network when mapped using different *Hmax* values with $T_{Hmax}^{total}(\alpha, \beta)$ as the optimization metric. The results show that the selection of *Hmax* is not critical if $T_{Hmax}^{total}(\alpha, \beta)$ is minimized. Notice that different *Hmax* values resulted in similar iteration times for smaller instances of the network. Figure 3.16 on the other hand shows that selection of *Hmax* plays a more significant role if $T_{Hmax}(\alpha, \beta)$ is the chosen optimization metric. Notice that for the sample network, mapping with *Hmax* = 100 leads to lower iteration times than mapping with *Hmax* = 50 or with *Hmax* = 200.

Figure 3.17 shows the folding digits selected for different *Hmax* values when $T_{Hmax}(\alpha, \beta)$ is the chosen criterion. The folded digits in each case are indicated by shaded areas. As an illustration, let us consider how different *Hmax* values affect the performance of the smallest network (a network with $H = 0$). Based on our approach, the optimal mapping for a network with no hidden neurons keeps 3 input and 2 output segment digits of the *VKNC* label intact. Notice that the dimensionality of the actual architecture is 5.

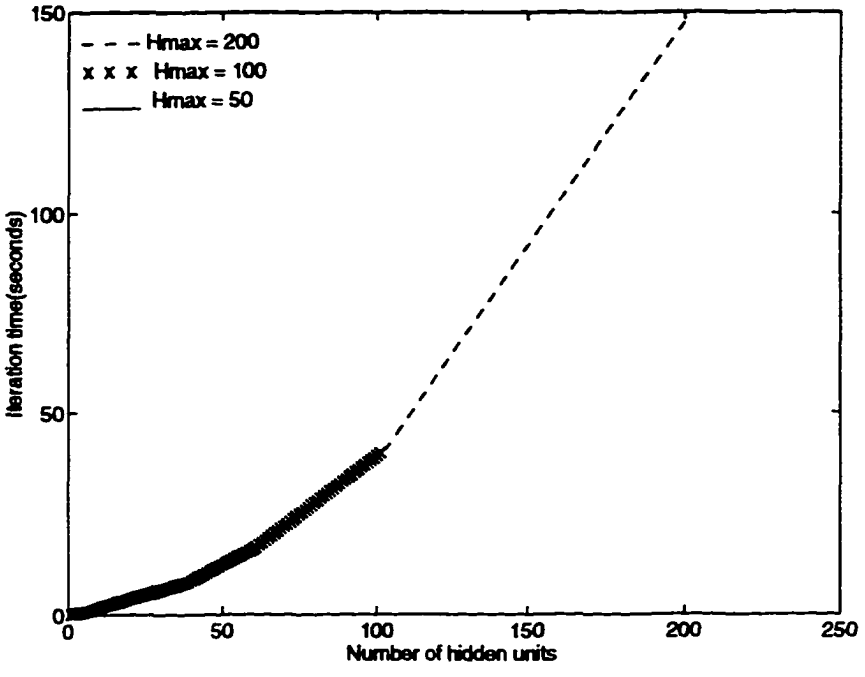

Figure 3.15: Simulation results of mapping with T_{Hmax}^{total} as the optimization metric with different *Hmax* values.

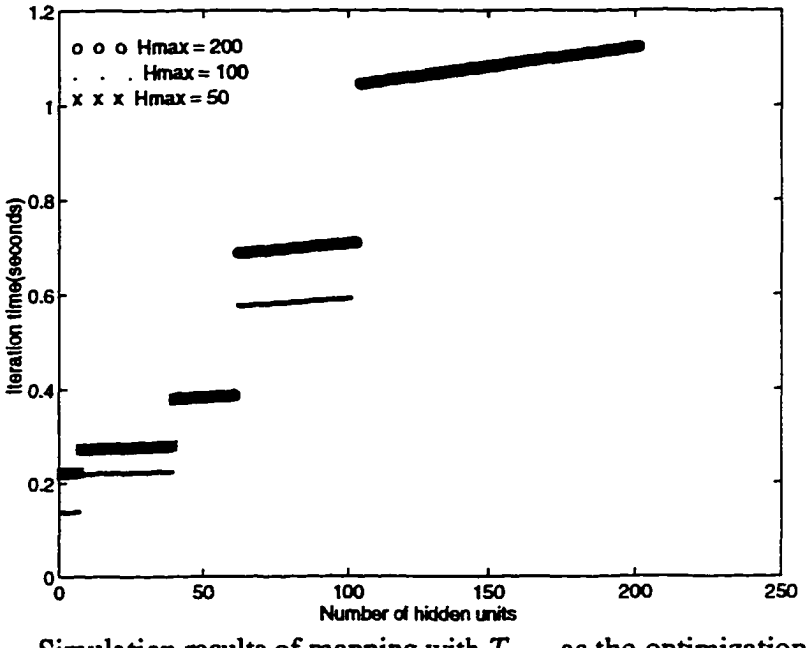

Figure 3.16: Simulation results of mapping with T_{Hmax} as the optimization metric with different *Hmax* values.

Hence, there is a total of 5 address digits. The mapping with *Hmax* = 100 also keeps these digits intact. However, the mapping with *Hmax* = 50 or *Hmax* = 200 chooses a different set of folding digits; i.e., they keep 2 input and 3 output digits. Hence, folding using the most significant input digit would result in different processor assignments for a network with no hidden units. This would clearly lead to higher iteration times. Figure 3.18 on the other hand shows that minimizing $T_{Hmax}^{total}(\alpha,\beta)$ with different *Hmax* values chooses the same folding digits. We can attribute this significant behavior to the fact that the optimization metric $T_{Hmax}^{total}(\alpha,\beta)$ takes into account the iteration times of all instances of the network. Thus clearly $T_{Hmax}^{total}(\alpha, \beta)$ is a better criterion than $T_{Hmax}(\alpha, \beta)$.

So far we have considered only the mapping of networks with up to *Hmax* hidden units. The final issue we need to address here is how to map the algorithm if the number of hidden units allocated during the training grows beyond the assumed upper bound, *Hmax*. Notice that in general we cannot estimate the ultimate number of the hidden units with high accuracy. Our approach is to apply the same mapping procedure for a network with *Hmax* hidden units to larger networks. Notice that once the folding process concludes, portions of the n_a-digit address of the parallel architecture are dedicated to input and output segments of the *VKNC* address. For larger networks, we fold the corresponding *VKNC* until its address matches the pattern obtained during the mapping of a network with *Hmax* hidden units. For instance, Figure 3.18 shows that after mapping a network with *Hmax* = 80, 3 digits have been allocated for the input segment of the *VKNC* address while 2 digits have been allocated to its output segment.

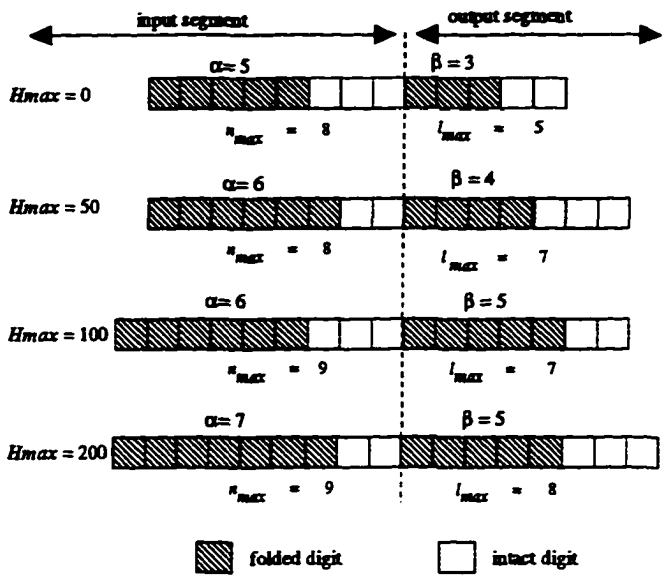

Figure 3.17: Folding digits with T_{Hmax} as the optimization metric for different *Hmax* values.

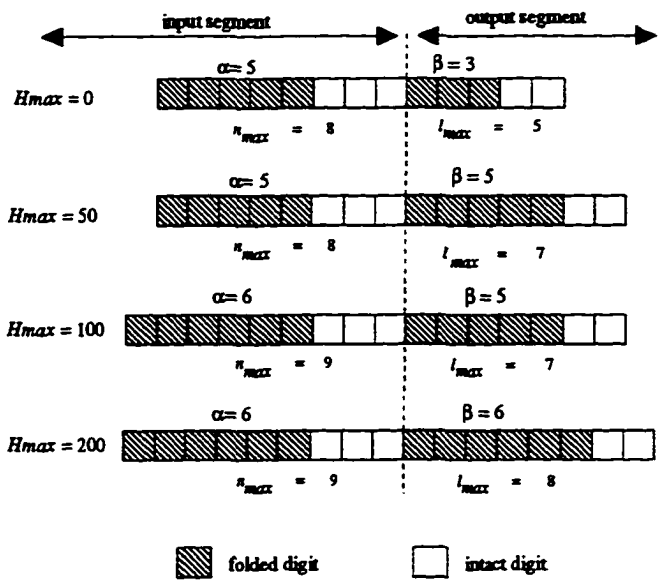

Figure 3.18: Folding digits with T_{Hmax}^{total} as the optimization metric for different *Hmax* values.

To examine the efficiency of the above approach, we mapped the network for the NETTalk application reported in [64] assuming *Hmax* = 40. Then, we applied the resulting mapping to larger networks (with up to 1000 hidden units). We compared iteration times obtained based on this mapping approach with the corresponding optimal values for different networks. The results of our simulations are depicted in Figures 3.19 and 3.20. These results show that our approach leads to near-optimal results even when the number of allocated hidden units grows beyond *Hmax*.

3.3 Mapping Adaptive Resonance Theory Networks

In this section we address the mapping of another unit-allocating *ANN*, the *Adaptive Resonance Theory* (*ART*) model. *ART* [17][31] neural networks are capable of adaptively categorizing an arbitrary sequence of (input) patterns into several clusters [33]. The clustering is typically performed based on a similarity measure [33]. For instance, all input patterns close to a particular vector (in Euclidean distance) are classified into the same group. *ART* was originally introduced by Grossberg [31] as a phenomena in human and animal cognitive information processing. This phenomena has led to the development of a series of unsupervised and supervised neural networks capable of pattern clustering and recognition. The resulting *ANN* models include *ART1* [18] which can categorize binary patterns, *ART2* [17] which can group both analog and binary patterns, *ARTMAP* [16] , a class of supervised neural networks, which is capable of category recognition and multi-dimensional maps, *FUZZY ARTMAP* [14], [15] , a modified version of *ARTMAP* which utilizes fuzzy neurons, and finally *ART-EMAP* [13] which is an *ANN* model capable of recognizing pattern classes after supervised and unsupervised learning.

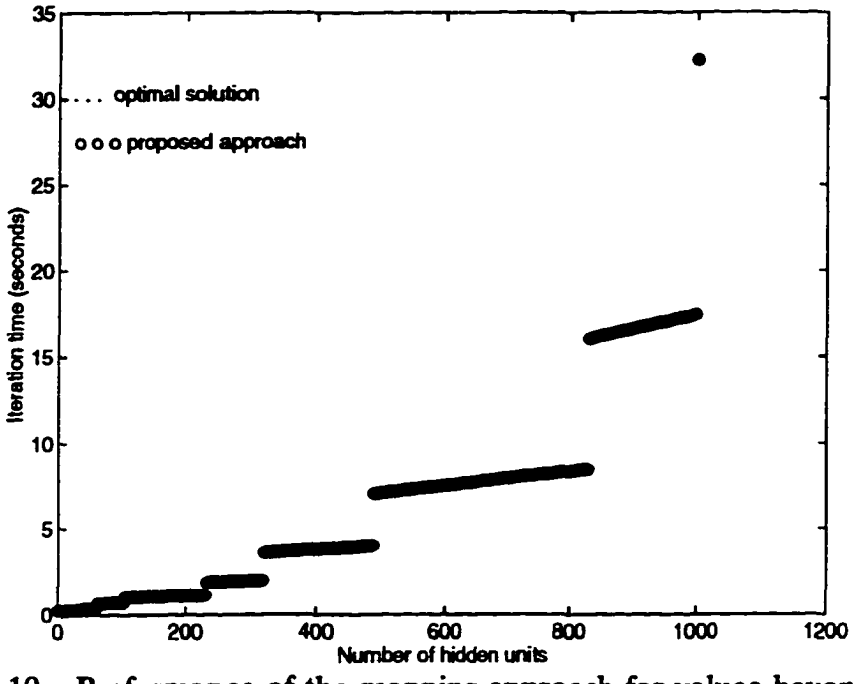

Figure 3.19: Performance of the mapping approach for values beyond *Hmax*, assuming T_{Hmax} as the optimization criterion and *Hmax* = 40.

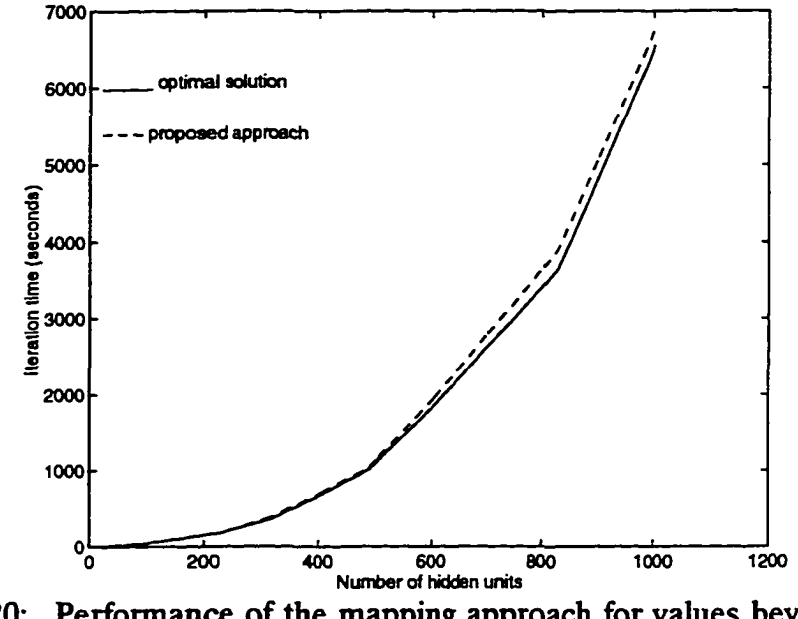

Figure 3.20: Performance of the mapping approach for values beyond *Hmax*, assuming T_{Hmax}^{total} as the optimization criterion and *Hmax* = 40.

Most variants of the *ART* model have been developed based on the original model, *ART1*. For instance, *ART2* is a variant of *ART1* which recognizes both binary and analog patterns. *ARTMAP* on the other hand incorporates several *ART1* modules. In this section we introduce a systematic mapping approach for highly efficient parallel simulation of the *ART1* training algorithm on *KNC* parallel architectures. Efficient simulation of other *ART* models can be derived from that of the *ART1* model. The simulation of the *ART1* model is a challenging problem since the corresponding network structure changes dynamically. We show that the general mapping approach we developed in Section 3.2 for efficient simulation of the *CC* learning (which is also a unit-allocating model) can be applied to the *ART1* model as well. We modify the mapping for the *ART1* model and examine its performance for a network running the NETTalk benchmark application.

3.3.1 Adaptive Resonance Theory 1 (*ART1*)

In this subsection we introduce the architecture and training algorithm of the *ART1* model. We adopt the abstraction of the *ANN* model presented in [33]. The *ART1* network consists of an input layer and an output layer as shown in Figure 3.21. Each output neuron represents a category or a cluster. We denote the output pattern for the k^{th} input pattern by an L-dimensional vector: $Y^k = (y_1^k, y_2^k, ..., y_L^k)^T$. It is assumed that any arbitrary input pattern belongs to only one cluster. Hence, each input pattern leads to only one *active* output unit, a unit whose output is 1. The number of clusters in the *ART1* model is not known apriori. Hence, the number of output units cannot be determined in advance. These units are allocated incrementally during the training until all input patterns can be classified based on some criterion which is discussed later. In fact, the *ART1* model is capable of

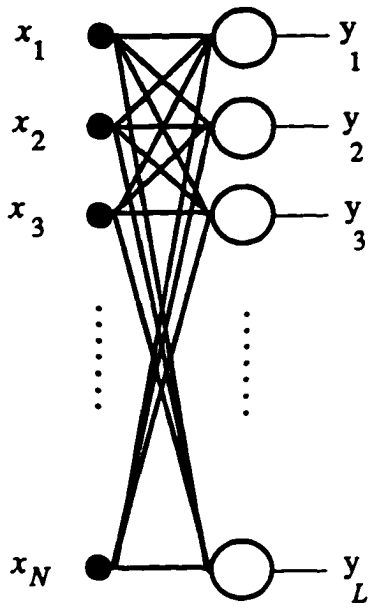

Figure 3.21: The *ART1* model.

accepting an infinite sequence of input patterns and allocating new clusters whenever necessary. Here, we use L to denote the number of output units that already exist in the network at a given time.

The input units of the *ART1* network are fully connected to its output units. The links between the input and output neurons are weighted. The k^{th} input pattern is expressed by an N-bit binary vector $X^k=(x_1^k, x_2^k,..., x_N^k)^T$. The weight between input unit i and output unit j is denoted by w_{ji}. The N-bit vector $W_j=(w_{j1}, w_{j2},..., w_{jN})^T$ is generally referred to as the *prototype* of cluster j. To classify each input pattern, say X^k, first the output of each existing output-layer unit, say j, is computed for the pattern as follows:

$$y_j = \frac{\sum_{j=1}^{N} w_{ij} x_j^k}{\|W_j\|^2} \tag{3.32}$$

where $\|W_j\|$ is the Euclidean norm of vector W_j. The output-layer unit with the highest output value is considered as the *winner*. Input pattern X^k is classified in the cluster associated with the winning output unit. Since only one winner can exist for each pattern, this process is refereed to as a *winner-takes-all* operation [33].

3.3.2 Training of the *ART1* Model

Each iteration of the *ART1* training algorithm involves examining the prototypes of the network for a given training pattern X^k. If the pattern X^k matches the prototype of an existing cluster according to a predetermined similarity test, then the pattern is added to the cluster. The cluster's prototype is adjusted to include the features of the new pattern. Otherwise, a new cluster is created with the new training pattern as its prototype. The similarity test consists of two major phases, a *winner-takes-all* phase and a *verification* phase. As we stated earlier, during the winner-takes-all phase, the output of each output-layer unit is computed for the pattern X^k. Then, the output-layer unit with the highest output is selected as the winner. During the verification phase, the prototype of the winning unit is examined to see if it matches pattern X^k well.

The verification phase consists of two steps. First, the prototype of the winning unit is examined to see if it satisfies the following inequality:

$$\frac{W_j^T X^k}{\|W_j\|^2} > \frac{\|X^k\|^2}{N} \tag{3.33}$$

If the inequality holds, it indicates that a significant fraction of the bits in the vector W_j and the bits in the vector X^k match [33]. The second test known as the *vigilance* test [33] involves testing the following inequality:

$$\frac{W_j^T X^k}{|X|^2} \geq \rho \tag{3.34}$$

where ρ ($0 < \rho < 1$) is a user defined parameter known as the *vigilance parameter*. If the inequality holds, it indicates that a significant fraction of the 1's in the pattern X^k and those in the prototype W_j match. The two steps are used to verify if the input pattern X^k can be classified in the cluster j (with the vector W_j as its prototype).

If the prototype of the winning output unit (W_j) passes both tests of the verification phase, then X^k is added to cluster j. In addition, the cluster's prototype is updated as follows:

$$W_j^{new} = W_j \wedge X^k \tag{3.35}$$

On the other hand, if the winning prototype passes the first test but fails the vigilance test, its corresponding output unit is deactivated (set to 0), and the output unit with the next highest output is selected as the new winner. This process is repeated until either a winner passes both verification tests or until all output units are deactivated. If no output unit passes the two tests, a new output unit (representing a new cluster) whose prototype is the pattern X^k is allocated. The above procedure can be formally presented as follows:

Algorithm 3.5:

/* *ART1* Training Algorithm */

Begin:

For (Each training pattern X^k) Do

1. *maximum* = 0 ;

2. For (Clusters $j = 1$ to L) Do

2.1 Compute: $y_j = \frac{W_j^T X^k}{\| W_j \|^2}$;

2.2 If ($y_j > maximum$)

$maximum = y_j$;

$winner = j$;

EndIf

EndFor

3.1 If ($\frac{W_{winner}^T X^k}{\| X \|^2} \geq \rho$ and $\frac{W_{winner}^T X^k}{\| W_{winner} \|^2} > \frac{\| X^k \|^2}{N}$)

$W_{winner}^{new} = W_{winner} \wedge X^k$;

3.2 ElseIf ($\frac{W_{winner}^T X^k}{\| W_{winner} \|^2} > \frac{\| X^k \|^2}{N}$ and Active units still exits)

$y_{winner} = 0$;

goto Step 2.2 ;

3.3 Else

$L = L + 1$;

$W_{L+1} = X^k$;

EndIf

EndFor

END

3.3.3 Mapping the *ART1* Model on *KNC*'s

The training algorithm of the *ART1* model incrementally adds output neurons until all unclassified patterns are properly labeled. The resulting structure is a two-layer network as shown by Figure 3.21. The simulation of the *ART1* model on parallel architectures has not been attempted before. As far as the mapping is concerned, the structure of the network

poses a challenging problem due to its dynamic nature. In principle, this mapping problem is similar to that of the *CC* algorithm we addressed in Section 3.2. Hence, we adopt a similar mapping methodology for parallel simulation of the *ART1* model. In particular, we perform the mapping assuming an upper bound on the number of output units (denoted by *Lmax*). We prove that our scheme minimizes the iteration time of the training algorithm for an *ART1* network with *Lmax* output units. Furthermore, through experimental results we show that the mapping leads to a very efficient simulation of other *ART1* networks with fewer than *Lmax* output units.

We can estimate the growing parameter of the *ART1* model better than that of the *CC* model. Clearly, the size of the *ART1* network depends on the number of possible groups into which the input patterns can be classified. For example, we know that the number of groups cannot exceed the number of input patterns. However, we should mention that *ART* models are capable of clustering an infinite stream of input data [33]. Under such circumstances obviously no limit can be set on the number of clusters unless we are interested only in certain clusters.

To utilize the parallel architecture efficiently, we first develop a parallel version of the training algorithm. We basically utilize the inherent parallelism of the original training algorithm, in particular Step 2 of Algorithm 3.5. Furthermore, we modify the algorithm by performing the vigilance test for all nodes concurrently. We believe that this would improve the overall efficiency of the algorithm. The modified algorithm is listed below:

Algorithm 3.6:

/* Parallel Training Algorithm for *ART1* */

Begin:

For (Each training patter X^k) Do

1. For (Clusters $j = 1$ to L) Parallel Do

1.1 Compute: $y_j = \frac{W_j^T X^k}{\| W_j \|^2}$;

1.2 If ($\frac{W_j^T X^k}{\| X \|^2} \geq \rho$ and $\frac{W_j^T X^k}{\| W_j \|^2} > \frac{\| X^k \|^2}{N}$)

$C\text{-}Flag_j = 1$; /* cluster j is a winning candidate */

else

$C\text{-}Flag_j = 0$; /* Cluster j is not a winning candidate */

EndIf

EndFor

2 For (Clusters $j = 1$ to L) Do

2.1 If ($y_j > maximum$ & $C\text{-}Flag_j == 1$)

$maximum = y_j$;

$winner = j$;

$active = 1$;

EndIf

2.2 If ($active = 1$)

$W_{winner}^{new} = W_{winner} \wedge X^k$;

2.3 Else

$L = L + 1$;

$W_{L+1} = X^k$;

EndIf

EndFor

END

The parallel *ART1* training algorithm can be modeled by a computational graph as shown in Figure 3.22. Nodes in this graph represent concurrent computations performed by the *ART1* neurons during one training iteration while edges represent the communication among adjacent neurons. The first layer of nodes in this graph corresponds to concurrent operations taking place during Step 1 of Algorithm 3.6. These include the computation of $\| W_j \|$, $\| X^k \|$, and $W_j^T X^k$. The second layer represents computations which take place in Step 2.1 of the algorithm. Clearly, the nodes in this graph represent computations which can be decomposed into atomic operations.

The computations in the first layer of the task graph precede those in the second layer. Hence the mapping problem can be simplified to that of mapping a bipartite graph such as that shown in Figure 3.23. We refer to this graph as $PART_{max}$. We assign n-digit ($lmax$-digit) k-ary addresses to nodes of $PART_{max}$ representing input-layer (output-layer) neurons of the *ART1* network, where:

$$n = \lceil \log_k N \rceil \ , \ lmax = \lceil \log_k Lmax \rceil \tag{3.36}$$

The mapping of the $PART_{max}$ graph onto the parallel architecture is similar to that of the PCC_{max} we introduced in Subsections 3.2.3 and 3.2.4. First, the computational graph $PART_{max}$ is mapped to a virtual *KNC* architecture called *VKNC*. Then, the *VKNC* is folded until its size is equal to that of the actual *KNC* architecture and a metric associated with the training time of an *ART1* network with *Lmax* output units is minimized. The

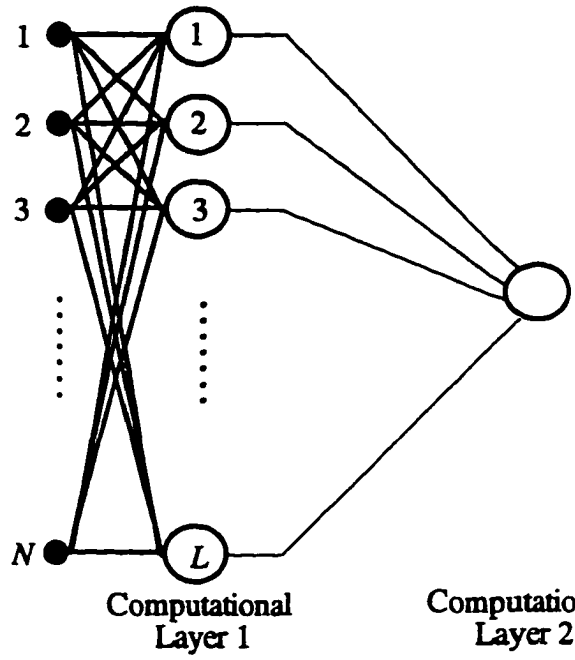

Figure 3.22: The *ART1* task graph.

optimization metric we use here is the sum of iteration times of the training algorithm for all instances of an *ART1* network with 1 through *Lmax* output units on the *VKNC* architecture. This metric is represented as $T^{total}_{Lmax}(\alpha, \beta)$ where

$$T^{total}_{Lmax}(\alpha, \beta) = \sum_{L=1}^{Lmax} T_L(\alpha, \beta) \qquad (3.37)$$

α and β denote the number of digits folded from input and output segments of the *VKNC* node address after $\alpha + \beta$ foldings. $T_{Lmax}(\alpha, \beta)$ denotes the iteration time of the largest instance of the *ART1* network after $\alpha + \beta$ foldings. We prove that the folding procedure minimizes the metric for a given assignment of tasks to processors of the *VKNC* architecture.

Theorem 3.8: If the training algorithm for an *ART1* network with *Lmax* output units and *n* input units is executed on a *KNC* architecture whose size matches that of the $VKNC^{\alpha+\beta}$

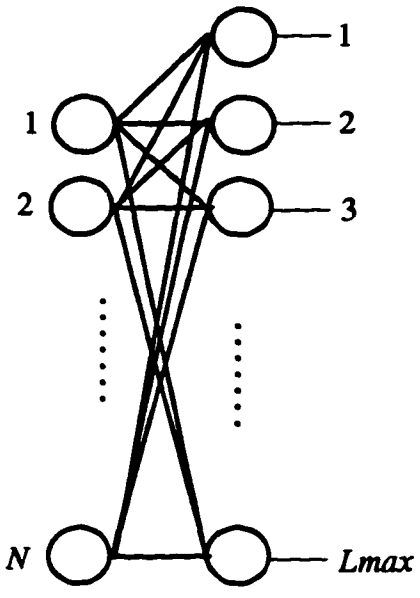

Figure 3.23: The *PARTmax* graph.

after α input and β output address digit foldings, then each iteration of algorithm will take:

$$T_L(\alpha,\beta) = max(k^{\beta-lmax-l}, 1)\left[(2k^{\alpha}+1)t_r + \left(\frac{1}{c_1}\min(l,lmax-\beta)+(n-\alpha)\right)(\lceil k/2\rceil t_c+2t_r)\right] \quad (3.38)$$

time units, where $c_1 = \max(k^{\beta-lmax+l},1)$.

Proof: Steps 1.1 and 1.3 of Algorithm 3.6 include the computation of $\| W_j \|$, $\| X^k \|$, and $W_j^T X^k$. These tasks can be performed as sum-of-products operations. According to Theorem 2.2, each of these operations takes:

$$\max(k^{\beta-lmax-l}, 1)\left[2k^{\alpha}t_r + (n-\alpha)(\lceil k/2\rceil t_c + 2t_r)\right] \quad (3.39)$$

time units. Step 2.1 of the algorithm involves the search for the minimum of L numbers. Based on our original task assignment, these numbers are uniformly distributed on a $\min(l,lmax-\beta)$-dimensional *KNC*. We utilize the binary spanning tree of the *KNC* architecture (for definition see Subsection 2.1.3) to compute the minimum. Basically, each node of the *BST* computes its local minimum and sends it to its parent. Then, the global

minimum is computed through a fan-in process (using an algorithm similar to Algorithm 2.2). Hence, the global minimum in Step 2.1 can be computed in

$$\min(l\ ,\ lmax-\beta)\ (\lceil k/2 \rceil t_c + 2t_r) + \max(\ k^{\beta\ -\ lmax\ -l},1)\ t_r \tag{3.40}$$

time units. Steps 2.2 and 2.3 take only $2t_r$ time units, which is relatively insignificant. Hence, the iteration time of the algorithm is given by:

$$T_L(\alpha,\beta) = max(k^{\beta-lmax-l}\ ,\ 1)\left[(2k^{\alpha}+1)t_r + \left(\frac{1}{c_1}\min(l,lmax-\beta)+(n-\alpha)\right)(\lceil k/2 \rceil t_c+2t_r)\right] \tag{3.41}$$

where $c_1 = \max(k^{\beta-lmax+l},1)$. □

Corollary 3.1: The iteration time of the training algorithm for the largest instance of an *ART1* network is given by:

$$T_{Lmax}(\alpha\ ,\ \beta) = k^{\beta}\left[(2\ k^{\alpha}\ +\ 1)\ t_r + \left(\ (lmax\ -\ \beta)k^{\ -\beta}\ +(n\ -\ \alpha)\right)(\lceil\ k/2\ \rceil t_c\ +\ 2t_r\ )\right] \tag{3.42}$$

Theorem 3.9: $T_{Lmax}(\alpha\ ,\ \beta)$ is a convex function with respect to α.

Proof: The second partial derivative of $T_{Lmax}(\alpha\ ,\ \beta)$ with respect to α is always positive: $\dfrac{\partial^2 T_{Lmax}(\alpha,\beta)}{\partial\alpha^2} = 6\ \ln k^2\ k^{\alpha+\beta}t_r$. □

The following corollary can be derived from Theorem 3.9 using a derivation similar to that utilized in the proof of Theorem 3.6.

Corollary 3.2: $T^{total}_{Lmax}(\alpha\ ,\ \beta)$ is a convex function with respect to α.

Define sets S_1 and S_2 as follows:

$$\begin{aligned} S_1\ &= \{\ L \mid 1 \le L \le Lmax\ ,\ l \le lmax\ -\ \beta\} \\ S_2\ &= \{\ L \mid 1 \le L \le Lmax\ ,\ l > lmax\ -\ \beta\} \end{aligned} \tag{3.43}$$

Theorem 3.10: The minimum value of $T_{Lmax}^{total}(\alpha, \beta)$ for a fixed β occurs at

$$\alpha = \begin{cases} \lceil B \rceil \text{ if } \max(0, n_f - lmax) < B < n \text{ and } T_{Lmax}^{total}(\lceil B \rceil, \beta) < T_{Lmax}^{total}(\lfloor B \rfloor, \beta) \\ \lfloor B \rfloor \text{ if } \max(0, n_f - lmax) < B < n \text{ and } T_{Lmax}^{total}(\lfloor B \rfloor, \beta) < T_{Lmax}^{total}(\lceil B \rceil, \beta) \\ \max(0, n_f - lmax) \text{ if } B \le \max(0, n_f - lmax) \\ n \text{ if } n \le B \end{cases} \tag{3.44}$$

where

$$B = \frac{Lmax\left(\lceil k/2 \rceil t_c + 2t_r\right)}{2 \ln k \, t_r \left(\| S_1 \| + \sum_{L \in S_2} k^{\beta - lmax + l} \right)} \tag{3.45}$$

and $\| S_1 \|$ is the cardinality of set S_1.

Proof: $T_{Lmax}^{total}(\alpha, \beta)$ is a convex function with respect to α. Hence, the minimum of the function for each β can be obtained from the critical points of the function or the end points of the range of α. The critical points of the function are obtained by setting the first partial derivative with respect to α to zero. This results in

$$\alpha = \frac{Lmax\left(\lceil k/2 \rceil t_c + 2t_r\right)}{2 \ln k \, t_r \left(\| S_1 \| + \sum_{L \in S_2} k^{\beta - lmax + l} \right)} \tag{3.46}$$

and $\| S_1 \|$ is the cardinality of set S_1. We denote the right hand side of the above equation by B. We are looking for an integer α, hence to find the minimum, we use $\lceil B \rceil$, $\lfloor B \rfloor$, or one of the end points of the range of α $(1, n)$, whichever leads to the lowest value when substituted in $T_{Lmax}^{total}(\alpha, \beta)$. This is formally represented by equation (3.47).

$$\alpha = \begin{cases} \lceil B \rceil \text{ if } \max(0,n_f - lmax) < B < n \text{ and } T_{Lmax}^{total}(\lceil B \rceil,\beta) < T_{Lmax}^{total}(\lfloor B \rfloor,\beta) \\ \lfloor B \rfloor \text{ if } \max(0,n_f - lmax) < B < n \text{ and } T_{Lmax}^{total}(\lfloor B \rfloor,\beta) < T_{Lmax}^{total}(\lceil B \rceil,\beta) \\ \max(0,n_f - lmax) \text{ if } B \le \max(0,n_f - lmax) \\ n \text{ if } n \le B \end{cases} \quad (3.47)$$

□

We use Theorem 3.10 to find the α and β which minimize the metric $T_{Lmax}^{total}(\alpha, \beta)$. The solution space for α and β for this optimization problem is similar to the closed sets shown in Figure 3.4. The only difference is that n_{max} in this figure should be replaced by n. We find the minimum of the function with respect to α in terms of β using Theorem 3.10. Clearly, this takes constant time. Then, we find the β which leads to the overall minimum. The time complexity of the overall search for the minimum is order $O(\log_k Lmax)$. This makes our mapping approach computationally very efficient.

3.3.4 Performance of the Proposed Mapping

In this section we evaluate the performance of the proposed mapping. In particular, we map the learning algorithm for an *ART1* network with *Lmax* output units on a *KNC* architecture using the proposed folding scheme. Then, we apply the resulting mapping to all instances of the network (networks with 1 through *Lmax* output units) and compute their iteration times. We then compare the iteration time of each instance of the network when mapped using our proposed scheme with its optimal iteration time. We wish to emphasize on the fact that our folding scheme minimizes $T_{Lmax}^{total}(\alpha, \beta)$ for an *ART1* network with *Lmax* output units. As we shall show, such a mapping leads to near-optimal results for

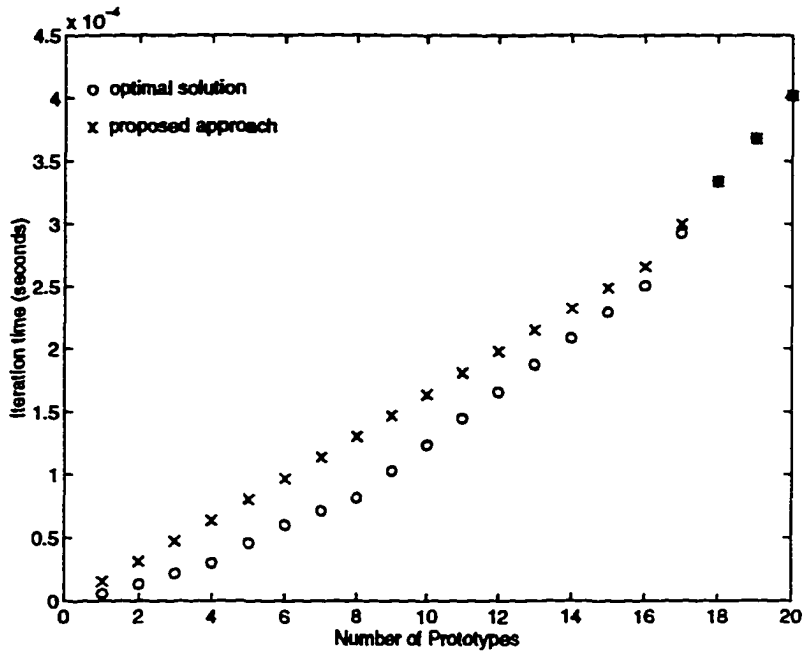

Figure 3.24: Simulation results for *Lmax* = 20.

other instances of the network with fewer than *Lmax* output units. The optimal iteration time for these instances of the network is computed through an exhaustive search for comparison purposes only. The results we report here are based on simulation of the network implementing the NETTalk application presented in [64]. In this case, there are 196 Input units, and 1019 training patterns. We assume that computation time (t_r) and communication time (t_c) are 0.01 μs and 4 μs, respectively. We also assume that the parallel *KNC* architecture has 16 processors.

For our simulations, we performed the folding so as to minimize $T_{Lmax}^{total}(\alpha, \beta)$ assuming a network with *Lmax* output units. We then used the resulting mapping to compute the iteration times for *ART1* networks with 1 through *Lmax* output units. We compared these results with their corresponding optimal values. We repeated our experiments with different *Lmax* values, namely *Lmax* = 20, 30, 40, and 100. Figures 3.24

through 3.27 show the results of our simulations. These results indicate that our mapping leads to near-optimal simulation of different instances of the *ART1* network. Insignificant deviations from optimal results appear only for small networks.

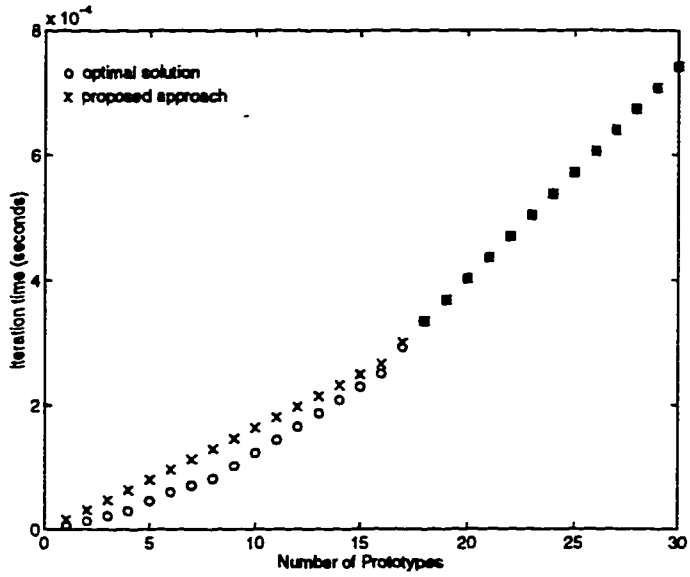

Figure 3.25: Simulation results for *Lmax* = 30.

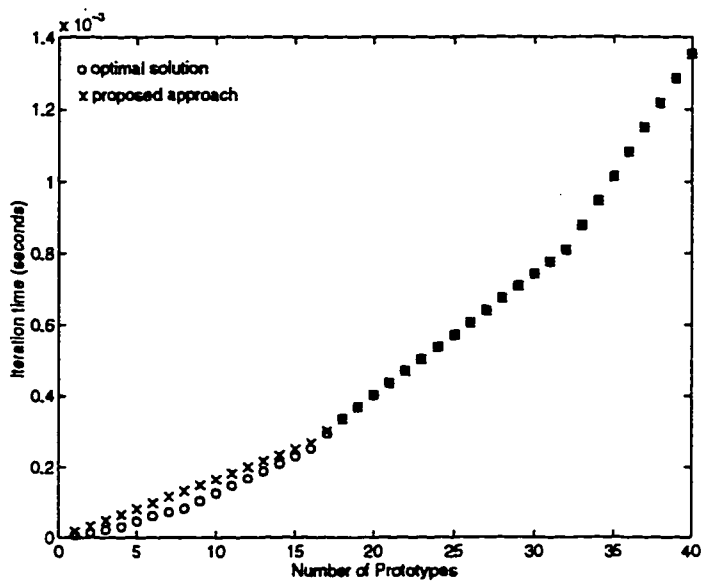

Figure 3.26: Simulation results for *Lmax* = 40.

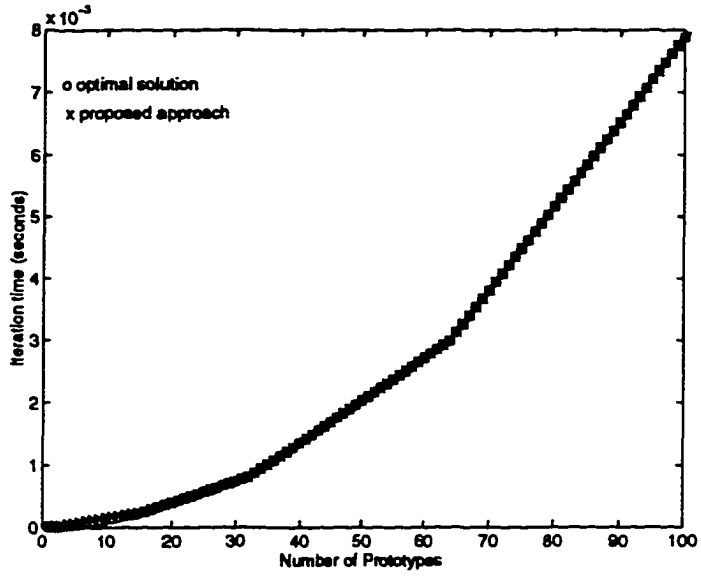

Figure 3.27: Simulation results for $Lmax = 100$.

CHAPTER 4

SUMMARY AND CONCLUSIONS

Artificial neural networks with their impressive processing power and inherent fault tolerance are viable candidates for solving many large scale scientific problems. These models offer a new processing paradigm which can be applied to problems intractable by conventional computing approaches. In this research we developed a formal methodology for efficient mapping of several contemporary *ANN* models on a popular class of parallel architectures. We considered parallel computers with k-ary n-cube topologies (*KNC*'s) since they encompass both mesh-connected and hypercube-based parallel systems. Many existing parallel architectures are based on these network topologies. Our mappings were designed to efficiently simulate *ANN* models of arbitrary sizes on *KNC* architectures.

Unlike earlier studies, we did not restrict our scope to a particular *ANN* model. Rather, we developed mapping schemes for several important classes of *ANN*'s. The classification we utilized grouped *ANN*'s based on similarities of their computational structures. Although specific implementations might vary from one model to another within a class, general mapping steps are similar for a given class of *ANN*'s. We presented a systematic mapping scheme for each class and showed how the mapping could be applied to specific *ANN*'s within each class. This feature of our study should have a significant impact on the study of *ANN*'s. With the availability of efficient implementations of a wide range of *ANN*'s at their disposal, researchers in the field can effectively study different aspects of neural processing in order to develop powerful problem solving tools. By

providing efficient implementation of a range of *ANN*'s for users, the practicality of these models in engineering problems is significantly enhanced.

One of the most important mapping criteria is to ensure that *ANN*'s are efficiently simulated on a given parallel architecture. The amount of parallelism achieved depends on the decomposition of the *ANN* model on the target architecture. To achieve an efficient simulation, it is essential to choose an appropriate granularity for partitioning the *ANN* model. The problem of determining the proper amount of parallelism for mapping an *ANN* model has not been addressed in most earlier studies. Most studies impose some restrictions on the size of the *ANN* model or the target architecture. Our unique mapping approach on the other hand systematically selected an appropriate degree of parallelism leading to a highly efficient realization of the *ANN* model on the host architecture. The scheme took into account several factors for determining the most suitable task granularity. These factors included the computational structure of the *ANN* model and the characteristics of the target *KNC* architecture, specifically its computation time for atomic arithmetic operations and per word communication time between adjacent nodes. If necessary, the scheme could utilize a subset of the processors of a given *KNC* architecture (referred to as *subcube*). In such cases, the simulation on a subcube of the target architecture resulted in the most efficient simulation.

In Chapter 2, we proposed a formal methodology for optimal implementation of the backpropagation and similar algorithms on *KNC*'s. The methodology was developed by generalizing the optimal mapping of a bipartite graph. Initially, the *FFANN* was mapped onto a virtual *KNC*. The extent of parallelism was such that simulation of the learning

phase on the virtual *KNC* was time optimal. Then, the virtual *KNC* was recursively folded until its dimension matched that of the physical architecture or a subcube thereof, depending on the physical size that provided the best execution time. A systematic folding process was developed to minimize execution time of each learning pass and to preserve the degree of redundancy. This mapping approach was very efficient since its computational complexity was a logarithmic function of the network size. We proved that our mapping methodology was time-optimal regardless of the structure of the *FFANN*.

In the same chapter, we showed that the methodology developed for *FFANN*'s could be applied to several other classes of *ANN*'s. In particular, we considered the mapping of *Radial Basis Function* (*RBF*) networks. We showed that the training algorithm for these networks had computational structures which were similar to that of the backpropagation algorithm. We considered both supervised and partially supervised training algorithms. We showed that the mapping of the fully supervised *RBF* was similar to that of a two-layer feedforward neural network trained with the backpropagation algorithm. We stated that the partially supervised scheme consisted of two steps. We introduced an efficient scheme for efficiently implementing the first step of the training algorithm. Initially, we provided a parallel version of the step. Then, we mapped the parallel version of the step onto a target *KNC*. Our implementation was based on partitioning the training set among processors of the *KNC* architecture. We proved that a uniform distribution of training patterns among processors of a given *KNC* architecture minimized the iteration time of the first step of the algorithm. Furthermore, we obtained the dimensionality of the subcube of the target *KNC* which resulted in the time-optimal execution of that step. We also showed that the second

step of the algorithm could be mapped as a two-layer feedforward network trained with backpropagation algorithm.

In Chapter 3, we addressed the mapping of an important class of *ANN*'s called unit-allocating neural networks. This class includes several important *ANN* models such as the Cascade Correlation learning algorithm and the Adaptive Resonance Theory 1 model. The common feature of these models is the dynamic nature of their architecture which grows during the learning phase. (Hardware implementation of these models are difficult due to their dynamic structure.) We investigated the problem of efficiently implementing these unit-allocating *ANN*'s on existing parallel systems. To our knowledge, this has not been previously attempted perhaps due to the dynamic nature of the network architecture.

We first presented a methodology for parallel implementation of the cascade correlation neural network learning technique on *KNC*'s. The method rendered efficient simulation of the algorithm through pipelining of several training patterns in parallel. Moreover, the efficiency of the implementation was enhanced by utilizing the inherent parallelism of the training algorithm. Both the output-unit and hidden-unit phases of the training were modeled using a computational task graph. A pipelined computational model called *PCC* was developed based on the task graph to accommodate the processing of several training patterns in parallel. This overcame the inefficiency of the original network due to its potentially large depth. We pointed out that the space complexity (memory requirements) of mapping the pipelined *CC* algorithm was increased by a constant factor at each processor with respect to the non-pipelined implementation of the algorithm.

The mapping involved implementation of the computational model of the pipelined algorithm, denoted by *PCC*, on an n_a-dimensional *KNC*. This consisted of two steps. First, the *PCC* model was mapped onto a virtual *KNC* (called *VKNC*) of compatible size. Since the number of hidden units was not known in advance, we considered an upper bound for this value (called *Hmax*) and used a *PCC* model developed based on a network with *Hmax* hidden units. The *VKNC* was then folded repeatedly until the dimensionality of the folded graph was less than or equal to n_a , depending on which size yielded minimum time. The folding process was designed to optimize a desired metric for a network with *Hmax* hidden units. We considered two optimization criteria, one represented the iteration time of the largest possible network and the other corresponded to the average iteration time of the algorithm for both training phases, calculated based on networks with 0 through *Hmax* hidden units. We showed that mapping based on either of the two criteria led to very efficient simulation of all instances of the network (but the smallest). We also examined the effect of *Hmax* on the performance of the simulated algorithm. We showed that the choice of *Hmax* was not critical if the sum of iteration times was to be optimized (assuming *Hmax* hidden units). The minimization of each metric (assuming *Hmax* hidden units) had computational complexity $O\left(\log_k(L + Hmax)\right)$, for a network with L output units. Based on the proposed mapping, task assignments for networks with 0 through *Hmax* hidden units were known apriori. Hence, no data migration or task rescheduling was needed as the number of hidden units grew.

In the same chapter, we used the parameters for a network implementing the benchmark application NETTalk to evaluate the performance of our mappings. We

presented experimental results which showed that our approach led to near-optimal results for networks with H hidden units where, $H \leq Hmax$. In addition, we showed that the proposed scheme led to very efficient simulation of the training algorithm even if the number of hidden units exceeded *Hmax*. We also examined the effect of *Hmax* choice on the mapping.

Also in Chapter 3, we addressed the mapping of a popular clustering network called Adaptive Resonance Theory (*ART*) . The training algorithm of the *ART1* model incrementally adds output neurons until all training patterns are properly classified. We showed that the mapping of such a unit-allocating model was very similar to that of the cascade correlation training algorithm. We first developed a parallel version of the training algorithm. We basically utilized the inherent parallelism of the original training algorithm. We modified one step of the original algorithm, namely the vigilance test, in order to improve its overall efficiency. We then modeled the parallel version of the algorithm by a bipartite task graph, denoted by *PARTmax*. The overall mapping was simplified to that of mapping *PARTmax* onto the target *KNC*. This task graph was implemented on the *KNC* architecture using the same approach applied to the task graph of the cascade correlation algorithm. We showed how an *ART1* model with certain number of output units could be optimally mapped on a *KNC* architecture. Through experimental results, we illustrated how this mapping could lead to very efficient simulation of other instances of the network with fewer output units. The proposed mapping was very efficient because of its logarithmic computational complexity.

There still remains several open problems related to mapping of *ANN*'s. One needs to explore the mapping of other classes of *ANN*'s which were not consider in this dissertation and determine if the general mapping schemes developed here could be adapted to them. For instance, we believe that the mapping we developed here for simulating feedforward neural networks can be applied to other *ANN* models whose computations can be captured by a bipartite task graph. In addition, similar mapping schemes for other classes of parallel architectures such as multiple bus systems need to be developed.

Further, the fault tolerance aspects of simulated *ANN*'s need to be studied. Artificial neural networks can inherently tolerate failure of neurons or neuron connections to some extent. Several studies on fault tolerance of neural networks appear in [4], [5],[6], [20], [63], [58], [70], and [76]. In these studies, several schemes were introduced to enhance the fault tolerance of *ANN*'s. These schemes mostly modified the training algorithm of an *ANN* model or provided some degree of redundancy to tolerate the failure of some neurons or neuron connections. However, for simulated *ANN*'s we should be concerned with the failure of processors or connections of the underlying parallel architecture rather than neuron or neuron connection failures. It might be possible to utilize the inherent fault tolerant capabilities of *ANN*'s to tolerate failure of processors. In particular, one needs to explore the possibility of modifying the training algorithm of *ANN*'s such that they can tolerate processor failures. For this, one needs to utilize robust training algorithms which can tolerate failure of multiple neurons such as the algorithms introduced in [4].

REFERENCES

[1] W. Allen and S. Saha, "Parallel Neural Network Simulation Using Backpropagation for the es-kit Environment," in *Proc. 1989 Conf. Hypercubes, Concurrent Computers and Applications*, 1989, pp. 1097-1102.

[2] J. Alspector, and R. B. Allen, A Neuromorphic VLSI Learning System, in *Advanced Research on VLSI*, (P. Losleben ed.), pp. 313-349, MIT Press, 1987.

[3] J. A. Anderson, "Neural Models for Cognitive Computations," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13, pp. 799-815, 1983.

[4] B. Arad, A. El-Amawy, "On Fault Tolerant Training of Feedforward Neural Networks", to appear in *Neural Networks*, 1997.

[5] B. Arad, A. El-Amawy, "Robust Fault Tolerant Training of Feedforward Neural Networks", *Proc. of 37th Midwest Symposium on Circuits and Systems*, August 1994.

[6] P. Bapat, "Design of fault Tolerant Feed-forward Neural Networks", *M.S. Thesis, Louisiana State University*, May 1994.

[7] A. G. Barto and P. Anandan, "Pattern Recognizing Stochastic Learning Automata," *IEEE Transactions on Systems, Man and Cybernetics*," SMC-15, pp. 360-375, 1985.

[8] G. Bellock and C. Rosenberg, "Network Learning on The Connection Machine," in *Proceedings 10th International Joint Conference on Artificial Intelligence*, Milan, Italy 1987.

[9] D. Berkey, *Calculus*, Saunders College Publishing, 1984.

[10] B. Bose, B. Broeg, Y. Kwon, and Y. Ashir, "Lee Distance and Topological Properties of *k*-ary *n*-cubes," *IEEE Trans. on Computers*, vol. 44, no. 8, pp. 1021-1030, August 1995.

[11] B. E. Boser, E. Sackinger, J. Bromley, Y. LeCun, and L. D. Jackel, "Hardware Requirements for Neural Network Pattern Classifier," *IEEE Micro*, vol. 12, pp. 32-40, 1992.

[12] D. S. Broomhead and D. Lowe, "Multivariate Functional Interpolation and Adaptive Networks," *Complex Systems*, 2, pp. 321-355, 1988.

[13] G. A. Carpenter, "ART-EMAP: A Neural Network Architecture for Object Recognition by Evidence Accumulation," *IEEE Transactions on Neural Networks*, vol. 6, pp. 805-819, July, 1995.

[14] G. A. Carpenter, S. Grossberg, N. Markuzon, J. Reynolds, and D. Rosen, "Fuzzy ARTMAP: A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps," *IEEE Transactions on Neural Networks*, vol. 3, pp. 698-713, September, 1992.

[15] G. A. Carpenter, S. Grossberg, and D. Rosen, "Fuzzy Art: Fast Stable Learning and Categorization of Analog Patterns by an Adaptive Resonance System," *Neural Networks*, vol. 4, pp. 759-771, 1991.

[16] G. A. Carpenter, S. Grossberg, and J. H. Reynolds, "ARTMAP: Supervised Real-Time learning and Classification of Nonstationary Data by a Self-Organizing Neural Networks," *Neural Networks*, vol. 4, pp. 565-588, 1991.

[17] G. A. Carpenter and S. Grossberg, "ART2: Self-Organization of Stable Category Recognition Codes for Analog Input Patterns," *Proceeding of IEEE International Conference on Neural Networks*, vol. II, pp. 727-736, San Diego, 1987.

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

[19] G. Chartrand and L. Lesniak, *Graphs & Digraphs*, Wadsworth & Brooks/Cole Advanced Books & Software, 1986.

[20] L. C. Chu and B. W. Wah, "Fault Tolerant Neural Networks with Hybrid Redundancy", *Proc. International Joint Conference on Neural Networks* , pp. II-639-649, San Diego, CA, June 1990.

[21] T. G. Clarkson, et al., "The pRAM: An Adaptive VLSI Chip," *IEEE Trans. on Neural Networks*, vol. 4, pp. 408-411, May 1993.

[22] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, (1973).

[23] M. Duranton and J. A. Sirat, "Learning on VLSI: A General Purpose Digital Neurochip," *International Conference on Neural Networks*, Washington DC, 1989.

[24] A. El-Amawy and P. Kulasinghe, "Algorithmic Mapping of Feedforward Neural Networks onto Multiple Bus Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 2, pp. 130 - 136, February 1997.

[25] S. E. Fahlman and C. Lebiere, "The Cascade Correlation Learning Architecture," in *Neural Information Processing Systems* 2, D. S. Touretzsky, ed. Morgan Kaufman, 1990, pp. 524-532.

[26] Y. Fujimoto, N. Fukuda, and T. Akabane, "Massively parallel Architectures for Large Scale Neural Network Simulations," *IEEE Trans. Neural Networks*, vol. 3, no. 6, pp. 876-887, November 1992.

[27] P. T. Gaughan and S. Yalamanchili, "A Performance Model of Pipelined k-ary n-cubes," *IEEE Trans. on Computers*, vol. 44, no. 8, pp. 1059-1063, August 1995

[28] J. Ghosh and K. Hwang, "Mapping Neural Networks onto Message-Passing Multicomputers," *J. Parallel and Distributed Computing*, vol. 6, pp. 291-330, Academic Press, 1989.

[29] J. Ghosh and K. Hwang, "Critical Issues in Mapping Neural Networks on Message-Passing multicomputers," *Int'l. Symp. on Computer Architecture*, pp. 3-11, ACM/IEEE, 1988.

[30] H. P. Graf and P. DeVegvar, A CMOS Implementation of Neural Network Model, in Advanced Research on VLSI, (P. Losleben ed.), pp. 351-367, MIT Press, 1987.

[31] S. Grossberg, "Adaptive Pattern Classification and Universal Recording: I. Parallel Development and Coding of Neural Feature Detectors, *Biological Cybernetics*, vol 23, pp. 121-134, 1976.

[32] D. Hammerstorm, "A VLSI Architecture for High-Performance, Low-Cost, On-Chip Learning," *International Joint Conference on Neural Networks*, vol. 2, pp. 537-543, 1990.

[33] M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, MIT Press, Cambridge, Mass, 1995.

[34] H. M. Hastings and S. Waner, "Neural Nets on The MPP," in *Frontiers of Massively Parallel Scientific Computations*, J. R. Fisher, Ed., NASA, July 1987.

[35] S. Haykin, *Neural Networks A Comprehensive Foundation*, Macmillan college Publishing Company, 1994.

[36] D. Hebb, *The Organization of Behavior*, Wiley, New York, 1949.

[37] A. Hwang, et. al., "A Two Level Pipeline RISC Processor Array for ANN," *International Joint Conference on Neural Networks*, pp. 137-140, 1990.

[38] K. Hwang and J. Ghosh, "Hypernet: A Communication-efficient Architecture for Constructing Massively Parallel Computers," *IEEE Trans. on computers*, vol. c-36, pp. 1450-1465, Dec. 1987.

[39] J. S. Kane and T. G. Kincaid, "Optoelectronic Winner-Take-All VLSI shunting neural network," *IEEE Trans. on Neural Networks*, vol. 6, pp. 1275-1279 , September 1995.

[40] S. Kollias and A. Stafylopatis, "Parallel implementations of the backpropagation learning algorithm based on network topology," In I. Pitas (Ed.), *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, Wiley, pp. 233-258, 1993.

[41] S. Kollias and D. Anastassiou, "An Adaptive Least Squares Algorithm for the Efficient Training of Artificial Neural Networks," *IEEE Trans. on Circuits and Systems*, pp. 1092-1101, August 1989.

[42] V. Kumar, S. Shekhar, and M. B. Amin, "A Scalable Parallel Formulation of The Backpropagation Algorithms for Hypercubes and Related Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1073-1090 October 1994.

[43] S.Y. Kung, *Digital Neural Networks*, PTR Prentice Hall, New Jersey, 1993.

[44] S.Y. Kung, J.N. Hwang, "A Unified Systolic Architecture for Artificial Neural Networks," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 358-387, 1989.

[45] S.Y. Kung, "Parallel Architectures for Artificial Neural Nets," *Proc. Int'l. Conf. on Systolic Arrays*, pp. 163-174, IEEE, 1988.

[46] J. L. Lansner, et al., "An Analog CMOS Chip Set for Neural Networks with Arbitrary Topologies," *IEEE Trans. on Neural Networks*, vol. 4, pp. 441-443, May 1993.

[47] C. Lehmann, et al., "A Generic Systolic Array Building Block for Neural Networks with On-Chip Learning," *IEEE Trans. on Neural Networks*, vol. 4, pp. 400-407, May 1993.

[48] P. H. W. Leong and M. A. Jabri, "A Low-power VLSI Arrhythmia Classifier," *IEEE Trans. on Neural Networks*, vol. 6, pp. 1435-1445 November 1995 1995.

[49] W. Lin, V. K. Prasanna, and K. W. Przytula, "Algorithmic Mapping of Neural Network Models onto Parallel SIMD Machines," *IEEE Trans. on Computers*, C-40, pp. 1390-1401, Dec. 1991.

[50] T. H. Madraswala, et. al., "A Reconfigurable ANN Architecture," *International Symposium on Circuits and Systems*, 1991.

[51] Q. M. Malluhi, M. A. Bayoumi, and T. R. N. Rao, "Efficient Mapping of ANN's on Hypercube Massively Parallel Machines," *IEEE Trans. on Computers*, vol. 44, no. 6, pp. 769-779, June 1995.

[52] W. S. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, vol 5, pp. 115-133, 1943.

[53] Micchelli, C. A. "Interpolation of Scattered Data: Distance and Conditionally Positive Definite Functions," *Constructive Approximation*, 2, 11 - 22, 1986.

[54] J. Moody and C. Darken, "Learning with Localized Receptive Fields," in Proceeding of the 1988 Connectionist Models Summer School (Pittsburgh, 1998), D. Touretzky, G. Hilton, and T. Sejnowski, eds., pp. 133 - 143. Morgan Kaufmann, San Mateo, CA, 1989.

[55] J. Moody and C. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, 1(2), 281 - 294, 1989.

[56] N. Morgan, et al., "The Ring Array Processor: A Mulitiprocessing Peripheral for Connectionist Applications," *Journal of Parallel and Distributed Computing* 14, pp. 248-259, 1992.

[57] R. Straub, D. Schwarz, and E. Schoneburg, "Simulation of Backpropagation Networks on Transputers," *Neurocomputing*, vol. 2, nos. 5 & 6, pp. 199 - 208, July 1991.

[58] C. Neti, M. H. Schneider and E. E. Young, "Maximally Fault Tolerant Neural Networks and Nonlinear Programming", *Proc. International Joint Conference on Neural Networks*, vol. 2, pp. 483-496, San Diego, CA, 1990.

[59] T. Nordstorm and B. Svensson, "Using and Designing Massively Parallel Computers for Artificial Neural Networks," *Journal of Parallel and Distributed Computing*, vol. 14, no. 3, pp. 260 - 285, 1992.

[60] E. Parzen, "On Estimation of Probability Density Function and Mode," *Annals of Mathematics and Statistics*, 33, 1065 - 1076, 1962.

[61] H. Paugam-Moisy, "Parallel Neural Computing Based on Network Duplicating," in I. Pitas (Ed.), *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, Wiley, pp. 305-340, 1993.

[62] A. Petrowski and H. Paugam-Moisy, "Parallel Neural Computation Based on Algebraic Partitioning," in I. Pitas (Ed.), *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, Wiley, pp. 259-304, 1993.

[63] T. Petsche and B. W. Dickinsom, "Trellis Codes, Receptive Fields, and Fault Tolerant, Self-Repairing Neural Networks," *IEEE Trans. on Neural Networks*, vol. 1, no. 2 pp. 154-166, June 1990.

[64] D. S. Phatak and I. Koren, "Connectivity and Performance Tradeoffs in the Cascade Correlation Learning Architecture," *IEEE Trans. on Neural Networks*, vol. 5, no. 6, pp. 930-934, November 1994.

[65] I. Pitas, *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, Wiley, 1993.

[66] T. Poggio and F. Girosi, "A theory of Networks for approximation and learning," *A. I. Memo 1140*, MIT, Cambridge, Mass, 1989.

[67] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. kung, "Neural Networks at Warp Speed: How We got 17 Million Connections Per Second," in *Proceedings of International Conference on Neural Networks*, San Diego, CA, June 1988.

[68] M. J. D. Powell, "Radial Basis functions for Multivariate Interpolation: A Review," in *Algorithms for the Approximation of Functions and Data*, J. C. Mason and M. G. Cox, eds. Clarendon Press, Oxford, England, 1987.

[69] D. E. Rumelhart, G. E. Hilton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing, Exploration in the Microstructure of Cognition, Volume 1: Foundations*, Cambridge, MA: MIT Press, pp. 318-364, 1986.

[70] C. H. Sequin and R. Clay, "Fault Tolerance in Feed-forward Artificial Neural Networks", *TR-90_031*, July 1990.

[71] N. B. Serbedzija, "Simulating Artificial Neural Networks on Parallel Architectures," *IEEE Computer*, pp. 56-63, March 1996.

[72] G. M. Shepherd and C. Koch, "Introduction to Synaptic Circuits," in *The Synaptic Organization of The Brain*, G. M. Shepherd (ed.), Oxford University Press, New York, pp. 3-31, 1990.

[73] R. K. Sitarman and N. K. Jha, "Optimal Design of Checks for Error Detection and Location in Fault-Tolerant Multiprocessor Systems," *IEEE Trans. on Computers*, vol. c-42, pp. 780-793, July 1993.

[74] M. A. Sivilotti et al., Real Time Visual Computations Using Analog CMOS Processing Arrays, *in Advanced Research on VLSI*, (P. Losleben ed.), pp. 295-312, MIT Press, 1987.

[75] D. F. Specht, "Probabilistic Neural Networks," *Neural Networks*, 3(1), pp. 109-118, 1990.

[76] M. Stevenson, R. Winter, B. Widrow, "Sensitivity of Feedforward Neural Networks to Weight Errors," *IEEE Trans. on Neural Networks*, vol. 1, no. 1, pp. 71-80, March 1990.

[77] H. S. Stone, *High-Performance Computer Architecture*, 3rd edition, Addison-Wesely Publishing Company, 1993.

[78] A. P. Thakkar, "Content-addressable, High-Density Memories Based on Neural Network models," *Technical report, Jet Propulsion Laboratory*, JPL D_4166, March 1987.

[79] A. Torralba, F. Colodro, E. Ibanez, and L. G. Franquelo , "Two digital circuits for a fully parallel stochastic neural network," *IEEE Trans. on Neural Networks*, vol. 6, pp. 1264-1268, September 1995.

[80] R. Ulrich, "SYNAPSE- A Neurocomputer that Synthesizes Neural Algorithms on Parallel Systolic Engine," *Journal of Parallel and Distributed Computing* 14, pp. 306-318, 1992.

[81] Anujan Varma, "Combinatorial Design of Bus-based Interconnection Structures", *Research report RC12550*, IBM Research Division, Sep. 1986.

[82] B. Vinnakota and N. K. Jha, "Diagnosability and Diagnosis of Algorithm-Based Fault-Tolerant Systems," *IEEE Trans. on Computers*, vol. c-42, pp. 924-937, August 1993.

[83] B.W. Wah and L. C. Chu, "Efficient Mapping of Neural Networks on Multicomputers," *Proc. Int'l Conf. on Parallel Proc.*, pp. I-234-241, Aug. 1990.

[84] T. Watanabe, et al., "A Single 1.5-V Digital Chip for a 10^6-Synapse Neural Network," *IEEE Trans. on Neural Networks*, vol. 4, pp. 387-393, May 1993.

[85] D. Wettschereck and T. Dietterich, "Improving the Performance of Radial Basis Function Networks by Learning Center Locations," in *Advances in Neural Information Processing Systems* 4, J. E. Moody, S. J. Hanson, and R. P. Lippmann, eds., pp. 1133-1140. Morgan Kaufmann, San Mateo, Calif. 1991.

[86] H. Yoon and J.H. Nang, "Multilayer Neural Networks on Distributed-memory Multiprocessors," in *Proc. Int. Conf. Neural Networks (IEEE/EEC)*, 1991, pp. 669-672.

VITA

Behnam Seyed Arad received his B.S. degree in Electrical Engineering with honors from the University of Massachusetts, Lowell in 1988. He received his M.S. degree in Electrical Engineering from Purdue University, West Lafayette Indiana in 1990. His research interests include high performance computer architecture, parallel processing, neural networks, and data communications.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Behnam Seyed Arad

Major Field: Electrical Engineering

Title of Dissertation: Efficient Mapping of Neural Network Models on a Class of Parallel Architectures

Approved:

A. El-Amawy
Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

John M. Tyler

Alex Ksevantyn

Chi Sik Lee

R. Vaidyanathan

Date of Examination:

April 4, 1997