DTIC FILE COPY

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY 2b.	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
4. AD-A222 146	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 90-010
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT Univ of Maryland 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION AFIT/CIA
6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6583
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) (UNCLASSIFIED)
Multiprocessor Realization of Neural Networks

12. PERSONAL AUTHOR(S)
Robert William Bennington

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
~~THESIS~~/DISSERTATION	FROM _____ TO _____	1990	334

16. SUPPLEMENTARY NOTATION
APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

DTIC ELECTE MAY 30 1990 S D D CS

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
ERNEST A. HAYGOOD, 1st Lt, USAF	(513) 255-2259	AFIT/CI

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"

MULTIPROCESSOR REALIZATION OF NEURAL NETWORKS

by

Robert William Bennington

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1990

Advisory Committee:

Professor Nicholas DeClaris, Chairman/Advisor
Professor Herbert Levitan
Professor Robert Newcomb
Professor Panos Ligomenides
Associate Professor Charles Silio Jr

ABSTRACT

Title of Dissertation: Multiprocessor Realization of Neural Networks

Robert William Bennington, Doctor of Philosophy, 1990

Dissertation directed by: Nicholas DeClaris, Professor,
Electrical Engineering Department

This research provides a foundation for implementing neural networks on multiprocessor systems in order to increase execution speeds and to accomodate more complex neural networks. The emphasis is on the use of affordable coarse grain multiprocessors to implement commerically available neural network simulators currently being run on single processor systems. A conceptual framework is presented based on the concepts of program decomposition, load balancing, communication overhead, and process synchronization. Four methodologies are then presented for optimizing execution times. A set of metrics is also introduced which make it possible to measure the performance enhancements over single processor systems, and analyze the effects of communications overhead, load balancing, and synchronization for various network decompositions.

The application of these four methodologies to two neural network simulators on a multiprocessor computer system is discussed in detail. They are illustrated with practical implementations of networks ranging in size from six to twenty thousand connections. Two of the methodologies, the Pipeline and Hybrid approaches, exhibit speedups approaching the possible upper limits.

The theoretical significant of this dissertation research is that it provides a basis for achieving efficient multiprocessor implmentation of highly and massive neural networks. Traditionally, neural network research and development requires a considerable amount of time be spent in repeatedly evaluating and modifying network architecture and algorithms. As such, the engineering value of dissertation is that the time required to repeatedly execute networks in research and development can be significantly reduced.

Dedication

Dedicated to

my loving wife, Patricia and

my three beautiful daughters

Kelly, Katy and Kerry

Acknowledgment

I would like to thank all my friends at work and at the university for all their support and encouragement during these past five years.

To Professor Nicholas DeClaris, my dissertation advisor, for his help in organizing and shaping the ideas presented in this dissertation and his constant encouragement and belief in me.

To Professors Robert Newcomb and Charles Silio for their practical advise and support during my years as a graduate student.

To Professor Herbert Levitan, for his friendship, advise, support, and especially for teaching me how to learn.

In addition, I would like to thank Mr Dennis Buck and Miss Colleen Goebel for their expert assistance in the preparation of this manuscript.

I would also like to express my thanks and love to my three lovely daughters, Kelly, Katy, and Kerry who, for most of their lives, have had to settle for a part-time daddy whose job it was to go to school.

Most of all, I would like to thank my wife, Patricia for her unwavering support, endless patience, and love, without which none of this would have been possible.

Bob Bennington

Beavercreek , Ohio

April 1990

Table of Contents

List of Figures

List of Tables

Chapter I
Introduction

Over the past five years there has been a resurgence in the research and development of artificial neural networks. A great deal of work has been done in attempts to have these biologically inspired networks mimic some of the properties of real neural networks. It is hoped that these artificial networks will be able to duplicate such properties as learning, massive parallelism, and fault tolerance [Caudill 1988a] [Ballard 1987] [D'Autrechy 1987] [Small 1983]. As such, researchers have applied neural networks to a variety of areas where conventional computer programs and expert systems fail, or show limited success. Such areas include character recognition [Burr 86], speech recognition [Burr 86] [Hwang 87] [Lippmann 87] [Peeling 86], text-to-speech recognition [Sejnowski 87], signal prediction [Lapedes 87], and protein structure analysis [Qian 88] [Levin 88].

Most, if not all, of the research being done in neural networks is aided by the use of a wide variety of computer systems running software simulations of these networks. Some of these simulations are being run on massively parallel processing computers such as the Connectionist Machine [Hillis 1984, Blelloch and Rosenberg 1987]. In such a machine, each node is represented by one

or more simple microprocessors with the interconnections between them mediated by a communication network, which is configured to mimic the connections of the neural network under investigation [Hillis 1984]. These types of systems use programming languages that have been developed to handle the special architecture and concurrent operating requirements of the system.

Other simulations are being run on specially developed neurocomputers [Aseo 1987] [Hecht-Nielsen 1988]. These neurocomputers are coprocessors that have been designed to implement one type of network (e.g. Hopfield, Avalanche, or Madaline), or a small number of related types (e.g. Backpropagation, Boltzmann, Cauchy, or Counterpropagation networks). They are connected to standard serial computers through a shared data bus or a peripheral interconnect, much like other external peripheral devices (e.g. a hard disk or printer). The standard computers act as the host, shuffling data into and out of the neurocomputer. The neural network, which the neurocomputer is implementing, does the actual data processing [Hecht-Neilsen 1988]. Unique software, developed specifically for the neurocomputers, is used for defining the network architecture and controlling the flow of data.

Because of the widespread availability of conventional serial computers, such as IBMs, SUNs, DECs, and VAX'es, most simulations are performed on such machines [Anderson 1983] [Klopf 1986] [Reggia 1985] [Fukushima 1983]. In general, researchers use these systems to run programs specifically written for their particular research needs [Fukushima 1983] or they make use of

general purpose neural network simulation programs such as MIRRORS I/II, Rochester Connectionist Simulator, Parallel Distributed Processor (PDP) Simulator, and P3 [D'Autrechy 1987] [Small 1983] [Goddard 1987] [McClelland & Rumelhart 1988]. These general purpose simulation programs are written in conventional programming languages such as LISP and C.

Unfortunately, computer usage in neural network research falls into two extreme catagories that each have their own unique problems. At one end of the spectrum are the single processor systems. With these systems, each node of the neural network is being processed sequentially by a single processor. Thus, one has the situation where the highly distributed architecture of a neural network is being modeled on a conventional computer system with one processor.

A second, more immediate problem on single processor systems, concerns processor speed. Although processor clock rates have been increasing, researchers have been steadily increasing the number of nodes and interconnections of networks, and applying more complex learning and transfer functions, in order to make the networks more adept at mimicking the cognitive functions of the brain. It is estimated that even relatively simple applications of these networks are likely to require thousands of processing units, and tens of thousands of interconnections, and that more sophisticated applications might require millions of processing units and billions of interconnections [Will 1987]. Thus, the overall result is that simulations are often extremely slow because of the sheer

size of the network and the large number of computations, and researchers are often limited in the size of the networks that can be practically simulated [Gilbert 1988].

At the other end of the spectrum are the fine grain parallel systems. These systems are configured such that one or more simple processing elements are used to represent a node in a neural network. Because of the number of processing elements involved, communications between nodes is typically via links with nearest neighbors and message passing to communicate with remote nodes. A good example of this type of system is the Connection Machine with its hypercube topology.

There are two main problems associated with these fine grain systems when implementing neural networks. While they are able to simulate large networks, they are fairly inflexible to changes in network architecture. That is to say, they require a lot of forethought on how the network should be implemented on the computer system, and it takes a long time to program the implementation. Thus, any gain in simulation speed can be easily offset by changes in network topology that often occur when experimenting with new and different networks.

The second problem area is in node communications. Because neural networks are highly connected, a high percentage of computer time is spent routing messages to the proper nodes. While efficient layout schemes alleviate

some of this problem, it is still a significant factor that slows down neural network simulations [Blelloch and Rosenberg, 1987].

Research Goal and Objectives

The research reported in this dissertation focuses on simulating neural networks on coarse grain computer systems to alleviate some of the problems exhibited by serial and massively parallel system simulations. Coarse grain systems can be characterized as having several powerful processors (compared to fine grain systems) with some local memory, and having a portion of main memory that can be shared by all the processors. Examples of such systems include the Sequent, Butterfly, and Masscomp computer systems.

The main goal of this dissertation research is to lay the foundation for the implementation of neural network technology on multiprocessor systems to increase simulation speeds and overall network size capabilities. To achieve this goal, several research objectives were established:

- Review & evaluate several neural network paradigms for multiprocessor implementation
- Review and evaluate available coarse grain computer systems
- Define a set of metrics for evaluating speed up of neural network simulations
- Evaluate the simularities in the architectures of neural networks and multiprocessor systems

- Examine and define those key concepts for the efficient implementation of a neural network on a multiprocessor system
- Develop methodologies for the implementation of neural network simulators on multiprocessor systems
- Apply the developed methodologies to commerically available software on a coarse grain computer system

Main Accomplishments

This dissertation's unique contributions to the advancement of neural network research are as follows:

- Development of four methodologies for implementing neural networks on multiprocessor systems: Layer, Cross-layer, Pipeline, and Hybrid Epoch Pattern
- Realization of average speedups of 1.66, utilizing the Pipeline method, and 2.0, utilizing the Hybrid Epoch-Pattern method, on a dual processor system implementation of several neural network simulators
- Development of a hybrid epoch-pattern training algorithm for multiprocessor implementation
- Development of a set of metrics that make it possible to measure performance enhancements over single processor systems and to analyze the effects of communication overhead and load balancing
- Successfully demonstrated the benefits of several methodologies for the immediate application to the R&D phases of neural network research

- Establishment of a basis for further work in multiprocessor implementation of neural networks

Organization

This dissertation is organized into eight chapters. Chapter two discusses background information on neural networks and various learning paradigms. Chapter three discusses material on coarse and fine grain multiprocessor systems. Chapter four details the challenge of implementing neural networks on multiprocessors. Chapter five discusses the four methodologies for implementing neural networks on coarse grain systems. Chapter six discusses the application of the methodologies to a Masscomp 5700 dual processor system and several neural network simulators. Chapter seven discusses my general conclusions and recommendations for the future work.

Chapter II

Artificial Neural Networks

The purpose of this chapter is to provide an introduction to artificial neural networks. It describes what artificial neural networks are and how they compare with their biological counterparts. The chapter details how artificial neural networks work, how they learn, and the different methods used to train them. It also describes several neural network architectures. The chapter closes with a detailed description of the backpropagation architecture as a means of showing how network topology, learning, and training interrelate.

Introduction

The study of neural networks and their properties is one approach being taken to model the gross structure of the brain, in an attempt to learn more about how the brain functions. These networks are called "neural" because they resemble and mimic the dense population of neurons and their axonal and synaptic processes that constitute the brain. As stated by Will [1987], "The principle assumption behind the neural network approach is that real intelligence is most readily achievable by mechanisms that closely resemble those mechanisms that exist in the human brain."

Whether or not these networks truly resemble, or operate in the same manner as the brain, is not of fundamental importance. What is important, is that the behavior exhibited by neural networks might enable researchers to develop theories and uncover basic principles that would explain the operation of the brain. From a more practical standpoint, much of the ongoing research is not so much concerned with how the brain operates per say, but rather how to build networks that exhibit such intelligent characteristics as fault tolerance, massive parallelism, and learning.

Many researchers feel that neural networks are very crude models of the brain, that pale in comparison to the capabilities of the brain. As such, they feel the word "neural" is inappropriate and prefer to describe these networks as parallel distributed processing systems, associative models, or connectionist networks, to emphasis their abstract nature [Rumelhart & McClelland 1986, Feldman & Ballard 1982, Fahlman & Hinton 1987, Will 1987]. Regardless of which term is used, they are generally accepted as being synonymous. We will use the term neural networks in this paper.

Biological Neurons

Despite the crudeness of neural networks, it is still important to understand the very basics of biological neurons. The following discussion, albeit brief, gives one a feel for the basis upon which neural networks are built.

The human brain is composed of an estimated 10^8 neurons. These neurons communicate throughout the brain and body via nerve fibers that

make an estimated 10^{15} interconnections called synapses. While neurons are biologically similar to other types of cells in the body, they possess the unique capabilities of receiving, processing, and transmitting electo-chemical signals. These signals are sent over neural pathways that make up the brain's communication system [Wasserman 89].

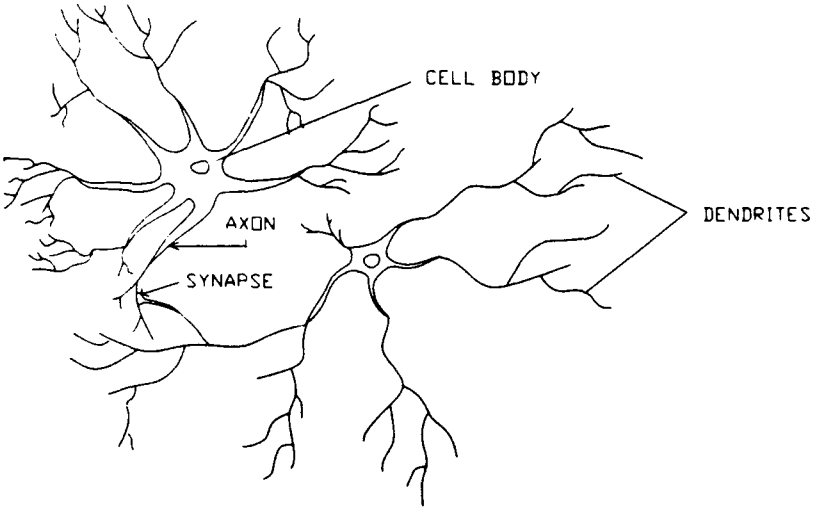


Figure 2-1

Biological Neuron

Figure 2-1 shows a typical neuron. It consists of three major parts; the cell body, the dendrites, and the axon. The dendrites extend from the cell body where they make connections, called synapses, with other neurons. The dendrites receive signals from other neurons via these synapses and conduct the incoming impulses to the cell body where they are summed. Some of the impulses tend to excite the cell while others try to inhibit the cell from firing. If the summation of all the inputs at the cell body exceeds a threshold, the cell fires, sending an electrical impulse down the axon to other neurons. Although seem-

ingly simplistic, these actions account for most of the known activity of the brain [Wasserman 89].

Artificial Neural Networks

Artificial neural networks consist of a collection of simple computational units usually referred to as processing elements, or simply, units. These units operate independently of one another and are completely self-sufficient [Hecht-Nielsen 1988]. Each one is capable of accepting input signals, performing some type of data processing, and sending signals to other units. Because these units operate independently, the network as a whole, is capable of making a large number of computations in parallel. And, because of their connectivity, signals from one unit can affect the output of other units, which ultimately affects the overall behavior of the network. These abilities enable neural networks to process large amounts of information simultaneously, much like the brain [Fahlman & Hinton 1987].

Another ability of neural networks is that of "learning", where learning is defined as the ability to modify the systems response to inputs over time. Each unit executes "rules" that tell it how to process information. Through the processing of information, the connections between the units are changed, adapting the network so as to assimilate new information. It is this adaptation that gives the network the ability to "learn".

Because neural networks can "learn", they possess the capability to rapidly process information in a non- algorithmic manner, unlike computers. As an example, suppose a neural network and a computer are both being used to classify objects according to shape. The computer would be programmed with one or more algorithms telling it how to go about classifying the object based on its shape. Because the computer has no capacity to "learn" no matter how many time it "sees", say, a circular shaped object, it processes what it sees by executing the same algorithms time after time.

This is not the case for neural networks. At first, the network goes through a series of training trials as it attempts to correctly classify the shape of the objects. During these trials, if the network incorrectly identifies an object, *a learning rule modifies the weights of the network's connections* (exactly how this is accomplished will be explained later). This modification enables the network to respond differently to the same shape the next time it is presented, and hopefully classify it correctly.

After being presented with each of the shapes several times, the learning rule has modified the weights in such a manner that the network can correctly classify each shape. Thus, the network has learned to "recognize" the shapes. At this point, the network is said to have "encoded" the various shapes in the connections of the network. Since no further weight modifications are necessary, the network no longer needs to execute the learning rule. Thus, while the computer must always execute its algorithms to classify the shapes, the neural network can immediately classify them without the further use of rules.

How Neural Networks Work

As mentioned above, a neural network consists of a collection of highly interconnected processing elements. Associated with each processing element is a number of input signals and a single output signal (figure 2-2). The inputs represent the outputs of other processing elements or "outside world" inputs to the unit. Associated with each input signal x_i is a weight w_i which is analogous to the synaptic strength of a real neuron. These weighted inputs are summed such that the effective total input, or Net = $\sum X_i W_i$.

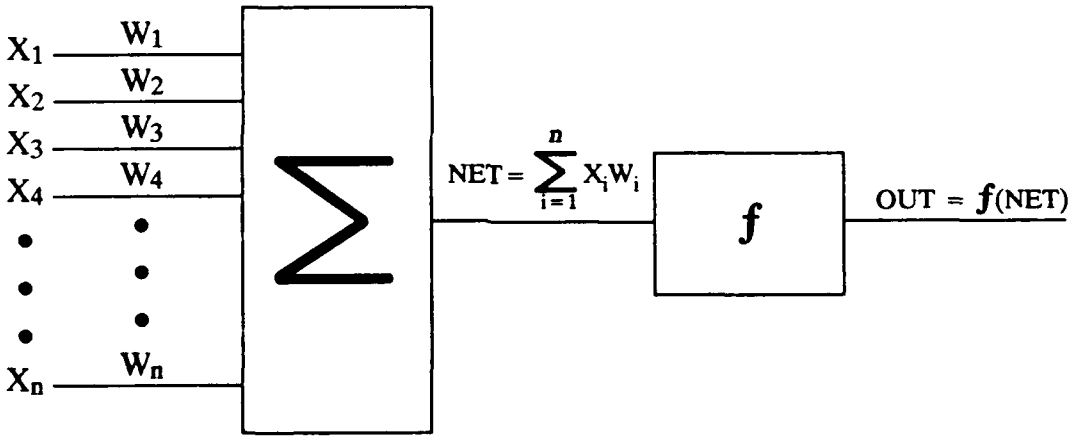


Figure 2-2

Single processing element with summation of input weighed inputs and activation rule f(net)

This summed input is further processed by the use of an activation rule. The activation rule is used to determine whether or not the unit will produce an output signal on the basis of the total input received by the unit. One of the

simplest activation rules is a fixed threshold. With this rule, if the summed input is greater than the threshold value the unit generates an output signal.

Thus, if Net > Threshold Out = 1

else Out = 0

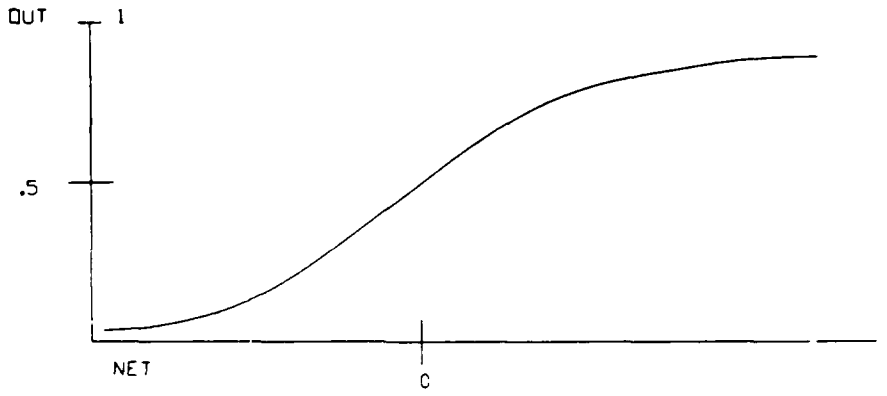


Figure 2-3

Sigmoidal Logistic Function

Another, very popular and useful activation rule is the sigmoid function. Here

$$\text{Out} = 1/(1+e^{-\text{Net}}) \text{ where Net} = \sum X_i W_i$$

Figure 2-3 shows a plot of the function. This function is often called the logistic or squashing function. It is popular because of its ability to compress, or squash, the range of Net so that the resulting Out lies between the values of zero and one. The function is also desirable because it provides a form of automatic gain control. For example, with small input signals (i.e. Net near zero) the slope of the sigmoid is steep. Thus small inputs produce high gain

outputs. However, as the magnitude of the summed inputs increases the slope of the sigmoid decreases, thus the gain also decreases. With this function, it is therefore possible for the network to accommodate large signals without saturation and small signals without excessive attenuation [Wasserman 89].

The signal output from the unit after being processed by an activation rule is often called the activation level of the processing element. As the name implies, it denotes the overall level of excitation or inhibition of the unit. Because the output of the processing element can be distributed to other processing elements, the value of the activation level becomes the input value to those other processing elements. In some networks, this activation level is a binary value (0 or 1), while in others, it is analog (e.g. any real number between 0 and 1) depending on the particular type of activation rule used [*Caudill 1988*, Lippmann 1987]. The activity level also acts as a "short term memory" for the processing element, allowing the network to store information (exactly how this is done will be explained below) [Fahlman & Hinton 1987].

Associated with each input signal is a weight. The weight represents the strength of the connection between two processing elements, or in other words, it denotes how much a change in one processing element will affect the other. The weights can be either positive or negative real numbers. Where negative reals denote an inhibitory influence, positive reals, denote an excitatory influence, and a value of zero, denotes no influence at all [Lippmann 1987, Rumelhart & McClelland 1986].

The weights are also the basis for producing the long term storage of information for the network. Information storage is accomplished by altering the weights associated with each connection, which in turn, changes the pattern of interconnections between processing elements [Fahlman & Hinton 1987]. Thus, one unique feature of neural networks is that information is distributed throughout the network in the weights of the connections.

The processing that alters the weights associated with each unit is determined by learning rules or algorithms. The learning rules specify how the weight of a connection is modified based on input values, activation level, and if present, "teaching" inputs [Rumelhart & McClelland 1986, Jones & Hoskins 1987, Hecht-Nielsen 1988].

One example of a learning rule is the delta rule. Under the delta rule, the change in weight between two processing elements is a function of the input from one processing unit and the difference, or delta, of the activation level achieved by a second unit and the desired activation level provided by a "teacher" [Caudill 1988, Rumelhart & McClelland 1986]. Thus,

$\Delta W_{ab} = n(T - O_b)(I_a)$ where,

n is a constant of proportionality representing the learning rate

T is the desired "teaching" output

O_b is the current output of the second unit

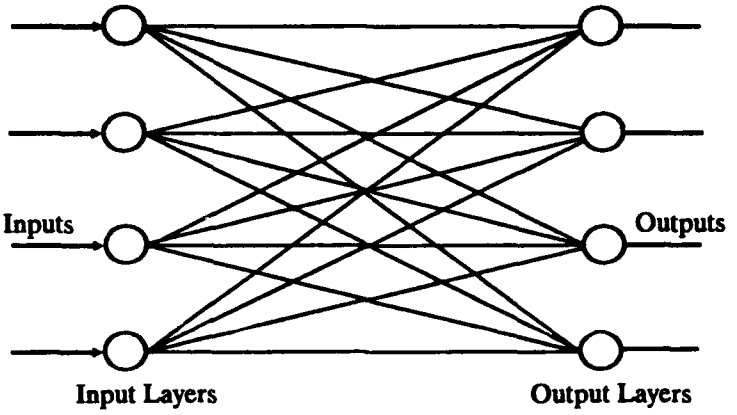
I_a is the input to the second unit from the first unit

The notation W_{ba} means the weight of the connection to b from a. Thus, directionality is implied. In some cases there could also be a W_{ab} connection.

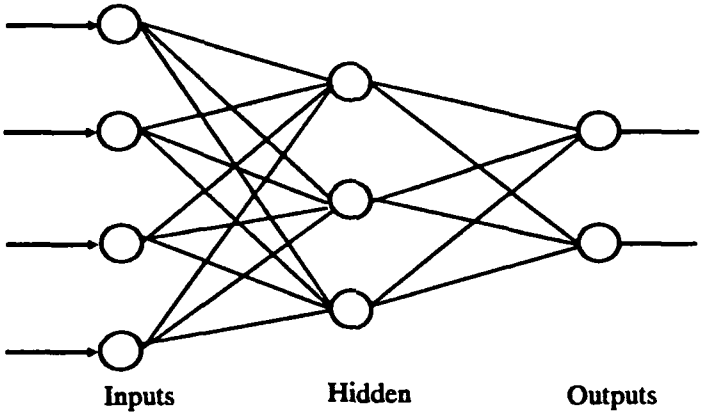
Network Architecture

The real power of these learning and activation rules comes from the interconnection of these processing elements to form networks. By connecting the outputs of various processing elements to the inputs of others, in a specific manner, groups, or layers of units are formed. The stacking and interconnecting of the layers in turn, forms the neural network.

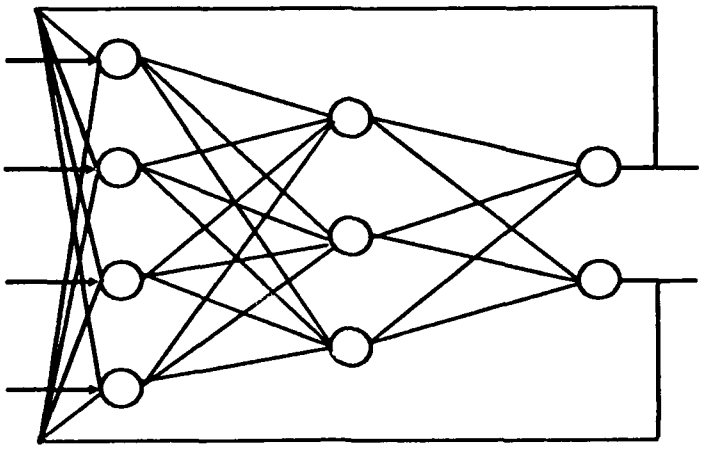
The simplest network consists of a single layer of processing elements as shown in figure 2-4a. Although the figure depicts two physical layers, the "layer" on the left only serves to distribute the inputs. Since the nodes in the layer perform no computations, the standard convention is to not consider them a layer of the network. In spite of this convention, those units that receive inputs directly from the "outside world" are often referred to as the input layer or layer 0 of the network. The layer that consists of units whose output is sent to the "outside world" is called the output layer. If a network has more than one layer, as shown in figure 2-4b, those additional layers are referred to as "hidden" layers. They are called this because they are "hidden" from the outside world by the input and output layers. The number of layers,



Single Layer Network (a)



2 Layer Network (b)



Recurrent Network (c)

Figure 2-4

Single and Multilayer Networks

and how they are connected to one another, determines the particular topology of the overall network [Fahlman & Hinton 1987, Caudill 1988].

Figure 2-4 depicts three common topologies. In figures 2-4a and b the only difference is in the number of layers. Note however, that in figure 2-4c there is a feedback path from the output layer units to the input layer units. This type of network is referred to as a recurrent network. The response of such networks is dynamic in that after applying an input, the resulting output is fed back to the input, modifying the net input signal. This action then causes the output to be recalculated and the process is repeated over and over. Because the network is designed to be stable these oscillations dampen out and the output converges to a constant value.

It is important to note here that the topology of the network alone does not uniquely identify a particular type of network. For example, in figure 2-4b the network could be either a backpropagation or a counterpropagation network. While the names are similar, the operation of the two networks is radically different. The backpropagation network uses a generalized delta rule learning algorithm, whereas the hidden layer in the counterpropagation network uses a Kohonen learning rule and the output layer a Grossberg learning rule [Hecht-Neilson 88].

I feel that to uniquely identify a particular network it is necessary to identify the transfer functions (i.e. learning and activation rules) used in the network as well as the topology. I call the combination of these two features, the

Popular Neural Network Architectures				
Name	Topology	Learning Rule(s)	Application	Comments
Adaptive Resonance Theory (ART)	Multilayer recurrent net	Bottom up match top down vigiliance	Pattern recognition	Very sophistciated
Avalanche (Grossberg)	Multilayer intralayer connections	Outstar Learning Equations	continous speech recognition	Can handle temporal or time varying sequences of spatial patterns
Backpropagation	Multilayer feedforward	Generalized delta rule (error minimization)	Adaptive control, speech synthesis	Most popular network
Boltzman Cauchy	Multilayer	Simulated annealing (random changes in weights)	Pattern recognition for sonar, radar	Simple net in which noise function is used to find global minima
Counter-Propagation	2 layer recurrent network	Uses self-organizing and Outstar Equations (Kohonen and Grossberg)	Image compression statistical analysis	Functions as a self-programmable look-up table
Self-Organizing Learning Map (Kohonen)	Single layer	Self-organizing unsupervised learning	Topology preserving mappings	Continually learning, very fast, require alot of training

Table 2-1

network architecture. Table 2-1 lists several popular neural network architectures.

Network Learning

The purpose of these transfer functions is to enable the network to learn how to process information. To accomplish this, the network is run through a series of "training" trials. These trials expose the network to a set of carefully selected inputs that teach the network some task. So for example, if the network is being trained to classify objects by shape, the training set might consist of one or more instances of all the shapes it must learn to classify, such as a circle, a square, a rectangle, and a triangle.

One of three basic types of training methods is used when teaching the network how to process information: supervised, graded, and unsupervised [Rumelhart & McClelland 1986, Lippmann 1987, Kohonen 1984]. The particular method employed is determined by the type of learning rule used by the network. For example, the first two methods are used with learning rules that require a "teaching" input. When no teaching input is required, the unsupervised method is used for training.

In supervised training, the network is given the training data as well as the desired output. After each trial, the learning rule compares the output of the output layer to the desired output (supplied by the trainer). If there is an output error, that is, a difference between the desired output and the actual

one, then the weights associated with the output layer are modified according the the learning rule. If the network has more than two layers, any output error is then "propagated" backwards to the hidden layer that serves as the input to the output layer (if the network only has two layers the only weights are those associated with the connections between the input and output layers, thus there would be no reason to propogate any error). This hidden layer then adapts it's weights and propogates the error back to the previous layer. This process continues until the weights between the input layer and the first hidden layer have been changed. The learning rule associated with each unit in each layer determines how much the weight will change, and in what direction (i.e. an increase or decrease in the weight) [Rumelhart & McClelland 1986].

Through successive iterations of the training data, the learning rule keeps changing the weights until the output error is reduced to a predefined, acceptable level. When this occurs the weights have converged to a single set of values (actually the weights oscillate around a single set of values. The magnitude of these oscillations depends on the magnitude of the acceptable output error). Thus, the network has found one set of weights that, when given any input, cause the activation of a set of units that will give the "proper" response. More abstractly, the network has learned how to process and organize information such that it answers correctly. An example of a network that uses this method is the Backpropagation network [Rumelhart & McClelland 1986].

A variation of the supervised training method is graded training. Because these two methods are so similar, researchers often do not distinguish between them, and refer to both as supervised training [Hecht-Nielsen 1988, Lippmann 1987]. In graded training, the network is given input data, but it is not given any output data. Instead of being told the proper response, the network is given a grade telling how well it performed [Hecht-Nielsen 1988]. For example, suppose a network is being taught to classify objects according to color. Suppose further, that the network is being taught to group all red objects together. If the network classifies a blue object as red, the network might be given a grade of 0, meaning that it is wrong. If the network classifies a red object as red, then it might be given a grade of 1, meaning that it is correct.

The network uses this grade as a "feedback input" to the network. In a manner similar to the one mentioned above for supervised training, the "feedback input" generates an "error signal" that is propagated back to the various layers of the network. The learning rule for the various units in the layers use this "error signal" to determine how the weights should be adjusted. Successive iterations of the training inputs are run until the network correctly classifies all of the inputs.

The network devised by Pazzani and Dyer [1987] is an example of a network that uses this training method. In their network, Pazzani and Dyer use the generalized delta rule as their learning rule. Like the delta rule described above, this rule requires a "teaching" input. In their particular study however, this teaching input does not tell the network what the desired output should be,

rather it tells it whether or not the response was correct. Thus, based on the correctness of the response an error output is generated, telling the various processing units the magnitude and direction of any weight changes.

In unsupervised training, the network is not given any type of feedback on its performance. In networks that use this type of training, the learning rule is a "clustering" algorithm. A clustering algorithm takes the first input and selects it as an exemplar (i.e. a model or template) for the first cluster. The next input is then compared to the first cluster exemplar for a match. A parameter called the vigilance, or threshold is used to determine how close an input and exemplar must be to be considered a match. If a match is made, the input is incorporated into the exemplar by adapting the weights of the cluster. If the input is not close enough for a match, it is used as the exemplar for a second cluster. This process is repeated over and over for all subsequent inputs [Lippman 1987]. Thus, each cluster represents a particular class of inputs with attributes different from the other clusters. Kohonen's Self-organizing Map [1983] and Carpenter and Grossberg's Adaptive Resonance Theory network [1982] are examples of networks using this method.

Once the training process is over, the learning rules can be disabled (depending on the specific application or theory to be tested) and the network given "real" data to process. The disabling of the rule fixes the weights, preventing the network from learning any new relationships. This allows the network to speed up its processing since it no longer has to execute any learning rules

and make adjustments to the weights. Such a network has two speeds: one with learning enabled, and one with learning disabled [Hecht-Neilsen 1988].

Backpropagation Learning Paradigm

With the information covered in the previous sections, one should be able to understand the various neural network architectures being researched. Because most new developments are more of an evolutionary process, an understanding of several basic architectures enables one to understand a large number of networks and follow their progress. While it is worthwhile to examine three or four architectures in detail, it is beyond the scope and purpose of this paper. However, to better understand what is to follow in later chapters, it is worthwhile to examine the *backpropagation network* paradigm in detail.

Background

Before the backpropagation algorithm there was no systematic, theoretical approach for training multilayer neural networks. Credit for the invention of the algorithm is shared amoung several researchers. Rumelhart, Hinton and Williams presented a clear and concise description of the algorithm in their 1986 paper. However, back in 1974 Paul Werbos had described the method in his Harvard Masters thesis. It was however, the 1986 paper that is in part responsible for the resurgance in interest in neural networks. It is estimated that apporximately 80% of the ongoing research involves the backpropagation algorithm in one form or another [Jackel 89].The backpropagation algorithm is

an error minimization technique used to adjust the weights in multilayer networks, enabling the network to learn. The algorthim can be applied to both recurrent and nonrecurrent networks (however the algorithm for recurrent networks is slightly different then what will be presented here). The algorithm itself is an extension of Widrow-Hoff's delta rule for single layer linear networks. For this reason, the backpropagation algorithm is often referred to as the generalized delta rule. It is a generalization of the delta rule because it is applied to multilayer networks and, as important, it can handle non-linear activation functions (unlike the delta rule).

Operation of Backpropagation Algorithm

This section is a summarization and review of the work presented in Rumelhart [1986]. The algorithm works as follows. Given a network with initially random weights, an input vector is applied to the network and the resulting output vector is calculated. The output vector is compared with a desired output vector, or target vector. The magnitude of the difference between the actual and the target vector is then used to generate an error function. This function is a metric for how well the network has been trained. The algorithm then specifies how to calculate the weight changes to perform a gradiant descent of the error function. The error function is then backpropagated through the network specifying how much change is required in each weight of the entire network.

Relating this back to the previously presented information, backpropagation denotes a specific architecture. Network topology is multilayer with it

being nonrecurrent in most applications. The activation rule is nonlinear and, as will be discussed, must be nondecreasing and differentiable. Supervised training is required since the algorithm requires a target output.

Expressing the explanation above mathmatically, we have an error function E which measures the closeness of the actual ouputs of the network o_{pj} and the target outputs t_{pj} for a series of patterns p on which the network is trained. Thus

$$E_p = \sum \frac{1}{2} (t_{pj} - o_{pj})^2 \qquad [1]$$

where o_{pj} is the jth component of the output pattern for the pth pattern and t_{pj} is the corresponding output component. The overall error of the network over all of the training patterns can then be expressed as

$$E = \sum E_p \qquad [2]$$

The desired effect is to make weight changes that will minimize this error function. To do this, one needs to select weight changes proportional to the negative of the derivative of the error function. Thus, we want

$$\Delta_p w_{ji} \propto - \frac{\partial E_p}{\partial w_{ji}} \qquad [3]$$

Before proceeding any further we need to define the net output and the activation function. The net output for pattern p can be defined as

$$net_{pj} = \sum w_{ji} o_{pi} \qquad [4]$$

where w_{ji} is the weight to j from i and o_{pi} is the output of node i due to pattern p.

The activation function is

$$o_{pj} = f_j(net_{pj}) \qquad [5]$$

where f is a differentiable and non-decreasing activation function

(e.g. the logistic function $1/(1 + e^{-net})$).

Now, applying the chain rule for differentiation to the error function [3] with respect to the summmed outputs

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ji}} \qquad [6]$$

from [3] $$\frac{\partial net_{pj}}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} o_{pk} = o_{pi} \qquad [7]$$

defining $$\delta_{pj} = - \frac{\partial E_p}{\partial net_{pj}} \qquad [8]$$

Equation 6 becomes $$\frac{\partial E_p}{\partial_{pj}} = - \delta_{pj} o_{pi} \qquad [9]$$

In turn, equation 3 becomes $$\Delta_p w_{ji} = \eta \delta_{pj} o_{pi} \qquad [10]$$

where η is a constant of proportionality.

This then clearly defines how the weight changes should be made to follow a gradiant descent in E.

One problem, however still remains. What should δ_{pj} be for each node in the network? While it is easy to determine for the output layer nodes, how does one determine the error for the hidden layer nodes? In other words, it is easy to assign a certain amount of "blame" to each node in the output layer since they directly contributed to the resulting error E_p. However, how does one determine the amount of "blame" that should be affixed to, say node x of hidden layer y of the network? This is exactly what the generalized delta rule is able to answer. Rumelhart et. al. used a single recursive computation of the δ's that enabled them to propagate the error backwards and assign a proportional amount of "blame" to each node of the network.

Going back to equation 8 then

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}}$$

applying the chain rule again with the change in error as a function of output of a node and the change in output as a function of the summed inputs results in

$$\delta_{pj} = -\frac{\partial E_p}{\partial o_{pj}} \quad \frac{\partial o_{pj}}{\partial net_{pj}} \qquad [11]$$

from equation 5

$$\frac{\partial o_{pj}}{\partial net_{pj}} = f'_j(net_{pj}) \qquad \text{i.e. the derivative of the activation function} \qquad [12]$$

Now for $\frac{\partial E_p}{\partial net_{pj}}$, have to consider, 1) nodes in the output layer and 2) those nodes in the hidden layer. From equation 1

$$\frac{\partial E_p}{\partial net_{pj}} = - (t_{pj} - o_{pj}) \qquad [13]$$

then for the output layer nodes

$$\delta_{pj} = (t_{pj} - o_{pj}) f'(net_{pj}) \qquad [14]$$

Now for the case where the nodes are in the hidden layer. Here again we can use the chain rule to write the first part of equation 11 as

$$\sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial o_{pj}} = \sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ki} o_{pi} = \sum_k \frac{\partial E_p}{\partial net_{pk}} w_{kj} = \qquad [15]$$

$$-\sum_k \delta_{pk} w_{kj}$$

Thus for hidden layer nodes

$$\delta_{pj} = f'_j(net_{pj}) \sum_k \delta_{pk} w_{kj} \qquad [16]$$

This means that the δ's for all the output nodes are computed then they are subsequently used to calculate the δ's for the hidden layer nodes.

Table 2-2 summarizes these results. Equation 1 specifies that the weight change for a node j should be proportional to the product of the error signal δ_{pj}, for that node and the output of a node i which sends its output to node j. If

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pi}$$

Equation 1

$$\delta_{pj} = (t_{pj} - o_{pj}) f'(net_{pj}) \quad \text{(for output layer)}$$

Equation 2

$$\delta_{pj} = f'_j(net_{pj}) \sum_k \delta_{pk} w_{kj} \quad \text{(for hidden layers)}$$

Equation 3

Table 2-2

Backpropagation Algorithm

node j happens to be in the output layer then the error signal (equation 2) is the product of the difference between the actual and target output multiplied by the derivative of the activation function. If node j is in a hidden layer, then the error signal (equation 3) is the product of the derivative of the activation function of node j multiplied by the sum of all the error functions of those nodes connected to node j times their weights.

Thus, the application of the backpropagation algorithm gives a simple method for adjusting weights no matter how many layers are in the network or how many nodes in a layer. The steps taken to execute the backpropagation algorithm are summerized as follows:

1. Apply a training pattern to the network.
2. Calculate the output value o_{pj} for each output node.
3. Calculate the δ's associated with each output layer node (eq. 14).
4. Backpropagate the δ's to the nodes in the previous layer (eq 16).
5. Keep backpropagating the δ's for any subsequent layers.
6. Calculate the weight changes for the training pattern (eq 10).
7. Repeat steps 1 - 6 for the next training pattern.
8. Repeat steps 1 - 7 until weights converge.

Summary

From the information in this chapter the reader should now have a basic understanding of neural networks; what they are, the relationship between the topology, the activation rules, and the learning rules, and how the learning rules are used to adjust weights, enabling the network to learn.

Chapter III

Multiprocessor Systems

Introduction

As in many areas of research, computers are an essential tool. For various reasons however, the best tool is not always the one used. This is true for neural network research. Most research in this area utilizes conventional computer systems to implement these highly distributed systems. Although successful, the overall result is slow simulations and size limitations. These two factors are especially detrimental when researching new network architectures because of the dynamic environment associated with the research phase (e.g. repeated simulations with minor changes to determine optimum topology).

There are better tools available, namely multiprocessor computer systems. Research in the computer science community has shown these concurrent processing systems to be an effective means of achieving serial super-computer performance [Hillis 84]. In some respects, they are even better than super-computers because of their affordability and growing popularity.

With this in mind, the purpose of this chapter is to provide an introduction to the fundamental concepts of multiprocessor computer systems. The first section gives a brief overview of multiprocessors and the characteristic features the possess. The next three sections discuss processor granularity, memory types, and the different types of parallel architectures and the communications networks required to support these systems. The last section of the chapter details the architecture of the Masscomp 5700 computer system.

Overview of Multiprocessor Systems

All modern computer systems exhibit some parallelism, whether it be at the arithmetic, instruction, program, or job level. Thus, whether or not a system is defined as a parallel system is somewhat subjective. We will use the term multiprocessors to refer to those parallel systems with more than one CPU, to distinguish them from those parallel systems with a single CPU that are capable of executing several instructions or computing several seperate data items simultaneously.

To better understand where multiprocessors fit in the plethora of computer systems, we can look at Flynn's taxonomy [Flynn 72]. His classification of parallel architectures is based on the relationship between the instruction and data streams that are processed during program execution. A stream is defined as a sequence of items (instruction or data) as executed or operated on by a

processor [Hockney 88]. Depending on whether the instruction or data streams are single or multiple, four broad classifications can be defined.

The first classification is Single Instruction stream Single Data stream (SISD). Computers in this category have one stream of instructions initiate a single operation which leads to a single data stream of logically related arguments and results. Most single CPU systems containing a single arithmetic unit capable of only scalar arithmetic fall into this class. Examples of such systems include most personal computers.

The second classification is Single Instruction stream Multiple Data stream (SIMD). Computers in this category still have a single stream of instructions, but they are vector instructions that initiate the execution of data on multiple processing units. Because each unit contains different data, there are multiple data streams. Computers in this class include the ILLIAC IV, and the Connection Machine. This category also includes those systems that have associative or content addressable memory processors. In these systems a number of stored data items can be processed concurrently by a single instruction. An example is the Goodyear STARAN system.

The third classification is Multiple Instruction stream Single Data stream (MISD). There is some dispute over the types of systems that fit into this category. This particular classification implies that multiple instructions are simultaneously operating on a single data stream. If one takes the viewpoint that in pipleine processors, each data item is processed by different in-

structions in different segments of a pipeline, then some pipeline computers fit this category [Hayes 78]. Others, such as Hockney [88] feel ther are no systems that fit this category and that pipeline systems such as the CRAY-1 are better classified as SIMD systems.

The fourth and final classification is Multiple Instruction stream Multiple Data stream (MIMD). In this class, the multiple instruction stream implies that there are multiple instruction processing units and thus, multiple data streams. Most types of multiprocessors fall into this category. While there are numerous examples, some worth noting are the 4 CPU CRAY- XMP, the BBN Butterfly, INMOS Transputer, Sequent Balance, Alliant FX/8, and MASSCOMP systems.

While Flynn's taxonomy provides a gross characterization, multiprocessor systems can be further characterized by several particular sets of features. The first set of features are the number and types of processors utilized by the system. The second set, the size and type of memory used. And the third set, the communication channels between the various processors. The paragraphs that follow briefly detail these features.

Processor Granularity

Another basis for the classification of multiprocessor systems in the number of processing units the system possesses and their computational power. This basis is called processor granularity of which there are two broad

categories; fine grain and coarse grain. Fine grain systems are defined as those machines that have a large number of relatively simple processors. Similarly, coarse grain systems are defined as those machines having a small number of powerful processors.

There is no definitive dividing line between coarse and fine grain systems. That is to say, that if a system has over 50 processors it is automatically called a fine grain system, or if a system utilizes 100 80286 processors it is called a coarse grain system. For some systems the classification is very clear, for others the classification is more subjective.

Fine Grain

Thinking Machine's Connection Machine is probably the best known fine grain computer system. It has over 65k single-bit processing units with 4K bits of memory per unit. Characteristic of fine grain systems is the small amount of memory associated with each unit. As such, there is not enough processor memory to enable each unit to execute any significant type of program. For this reason, most fine grain systems operated as SIMD machines.

Most fine grain systems also have custom designed topologies to make them effecient at solving specific types of problems such as matrix calculations. The Connection Machine however, is somewhat of an exception for it has a flexible topology. It uses a complex switching network which provides programmable connections between any two processors. These connections can be

changed dynamically during program execution. This make this particular system more of a general purpose fine grain system.

Because of the large number of processors required in fine grain systems, some systems use custom designed processors. These processors are designed specifically for parallel processing. Two well known examples utilizing custom processors are, the INMOS Transputer and the NCUBE Hypercube.

Coarse Grain

At the other end of the spectrum are the coarse grain systems, which consist of a small number of more powerful processors. Coarse grain systems typically employ conventional sequential processors, such as those found in personal computers and work stations. The Motorola 68000 series and Intels 80x86 series are two such examples. Examples of coarse grain systems include the Masscomp 5700, Alliant FX/8 and the Cray-XMP.

The relatively small number of processors associated with coarse grain systems enable most such machines to have enough processor associated memory to store individual programs. As such, these sytems are capable of MIMD operations. This provides a great deal of flexability over SIMD machines and allows the processors to operated asynchonously. There are times however, that the processors need to interact if these systems are working on a common problem. Thus, with such systems, synchronization or "rendevous" techniques play an important role in the systems overall performance

(the importance of synchronization and other key issues will be discussed in a later chapter).

Memory Types

A second factor used to characterize multiprocessor systems is the type of memory it utlizes. Memory can be classified as either shared or distributed. Shared memory, also called global, is that memory that can be directly accessed by any of the system's processors. Distributed, or local memory, is that memory that can only be accessed by a single processor. It should be noted that cache memory is typically not consided as part of a multiprocessor's local memory.

Another set of terminology used to refer to shared and distributed memory is that of coupling. If a system is referred to as being tightly coupled, it means that the sytems utilizes shared memory. If the sytem is referred to as loosely coupled, it means the system utilizes distributed memory.

The granularity of a system somewhat determines the type of memory a multiprocessor system possesses. Fine grain systems are typically characterized as having local memory. The reason being, is that contention for shared memory resouces becomes unacceptable when the number of processors is greater than thirty or so (numbers vary depending on the type of communications used). Systems with local memories do not have to worry about memory contention since each processor has its own. There are tradeoffs however, in

that communications between processors with local memory is more complicated. Communication channels must be implemented so data can be passed back and forth. Several examples of system with local memory include the Connection Machine, Inmos Transputer, and NCUBE's Hypercube.

Characteristic of coarse grain systems is shared memory. As alluded to above, one of the advantages of shared memory is that of interprocessor communications. With a global memory, results from each processor and messages can be stored in predetermined memory locations so that other processors can access the information. Thus, dedicated interprocessor communication channels are not required. One other nice feature of shared memory is that multiple copies of common data are not needed for each processor to execute a computation. Examples of shared memory machines include the Sequent Balance, Masscomp 5700 series, and the Alliant FX/8 systems.

The main drawback of shared memory however, is that the processors must contend with one another for the memory resources. If the number of processors is small, the contention can be handled very well and there is little or no degradation in system performance. However, as the number of processors increases, contention does start to degrade system performance until it reaches a point where shared memory is no longer viable. This point varies amoung different systems depending on the type of architecture and communication networks it employes. In fact, some of the various coarse grain ar-

chitectures and communications networks were developed to address this very problem.

There are some systems that, depending on ones frame of reference, fit into either the shared or distributed memory catagories. Two notable examples are IBM's RP3 and the BBN Butterfly. Both systems are classified as fine grain systems because of the large number of proccessors they utilize (513 for the RP3 and 256 for the Butterfly). They possess banks of memory that, while physically located with each processor, can be accessed by any processor via sophisticated switching networks. Thus, if one considers all the memory banks as a whole, even though they are physically seperated, these systems can be classified as having shared memory. If however, each bank of memory is considered a seperate entity because of its location, then the systems can be classified as having distributed memories.

Multiprocessor Architectures and Communication Networks

Figure 3-1 depicts the two types of processor architectures described in the last two sections. Figure 3-1a illustrates a generic coarse grain architecture with its shared memory and communications network, while figure 3-1b illustrates a generic fine grain system with its distributed memory. How these processors communicatewith one another and their memory is dictated by the communications, or interconnection, network.

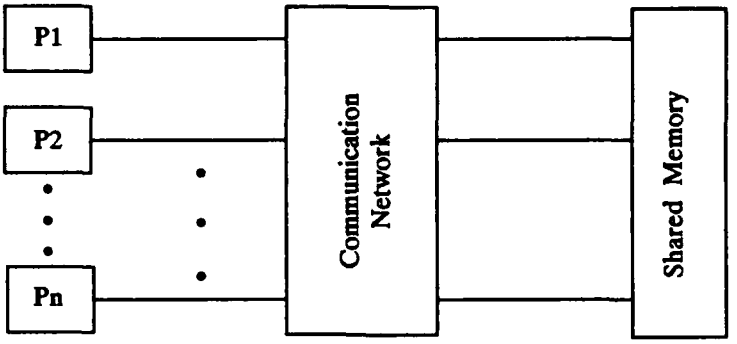
With multiprocessor systems, the communications network can take on a number of topologies as shown in figure 3-2a. Each node in the figure represents the combination of the processor and memory, often referred to as the computing element or processing unit. The lines between the nodes denote how the communications network is set up, visually depicting the links between the processing elements.

Because the communication network is crucial to the effecient operation of the system as a whole, it is worthwhile to describe some of the various types. Representative of the many interconnection schemes are:

- Shared Bus
- Crossbar connected
- Multistage switching
- Hypercubes

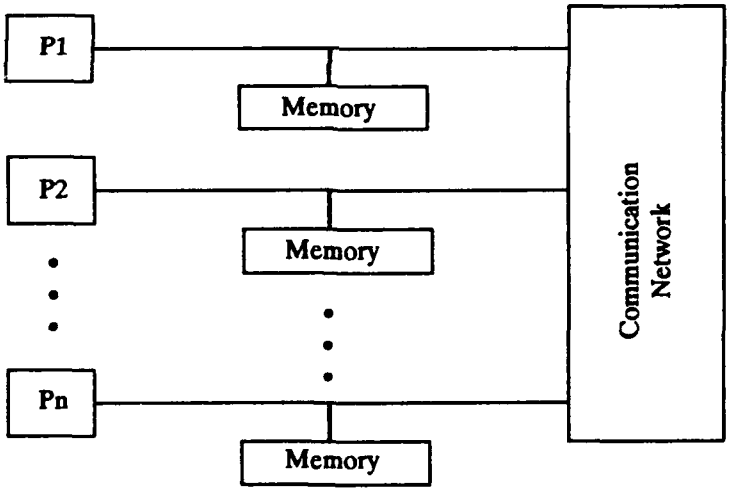
Shared Bus

The most common communication network used in coarse grain systems is the shared bus. Figure 3-3 depicts the shared, or common bus system. Associated with each processor is a cache and possiblly some small amount of local memory. The cache and local memory enable the processors to reduce their use of the bus, thus limiting potential conflicts with other processors. The bus provides temporary links between the processors via the shared memory. Examples of systems using shared buses are Masscomp, the Sequent Balance and the Alliant FX/8.



(a)

Coarse Grain System

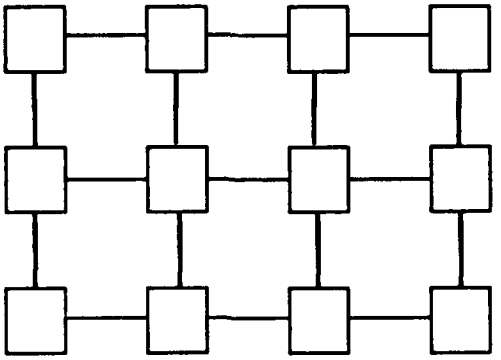


(b)

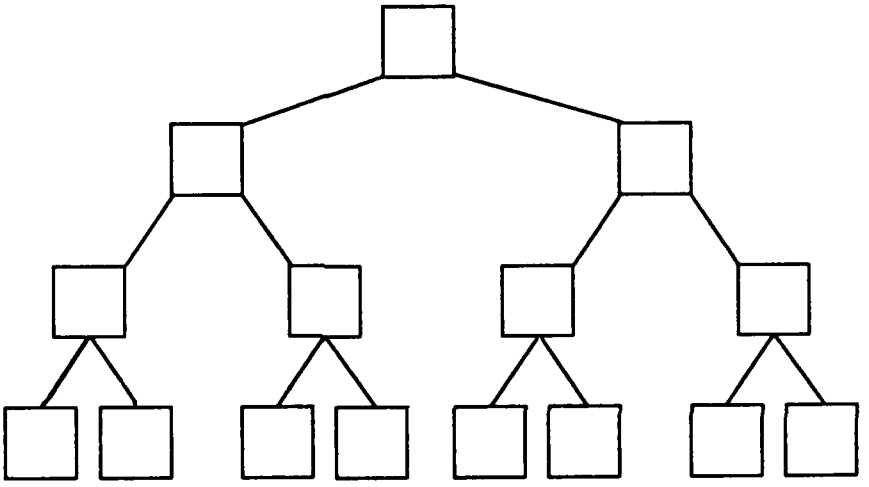
Fine Grain System

Figure 3-1

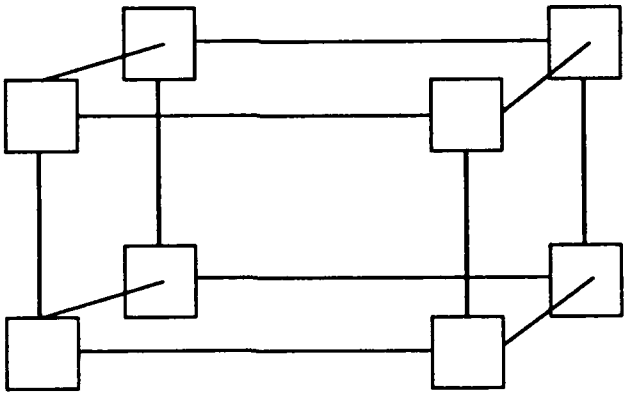
Coarse and Fine Grain Multiprocessor Architectures



(a) Mesh



(b) Heiarchical



(c) Cube

Figure 3-2
Common communications network topologies

The shared bus is very similar to the buses used in most sequential computer systems. It has interrupt lines, master slave relationship, and bus allocation procedures. And, like most buses, because it is a shared resource, it is the main bottleneck in effecient performance.

Crucial to the effectiveness of the bus oriented systems are the cache and local memory (if any). They are used to shorten the effective memory cycle and reduce the use of the bus so that one processor does not slow down another through the bus interface. With effective use of the cache and local memory, bus access can be significantly reduced thereby allowing more processors to share the bus at a given level of contention. Even with the use of high bandwidth buses and fast memory however, the number of processors that can be supported is usually between 20 to 30 [Stone 88].

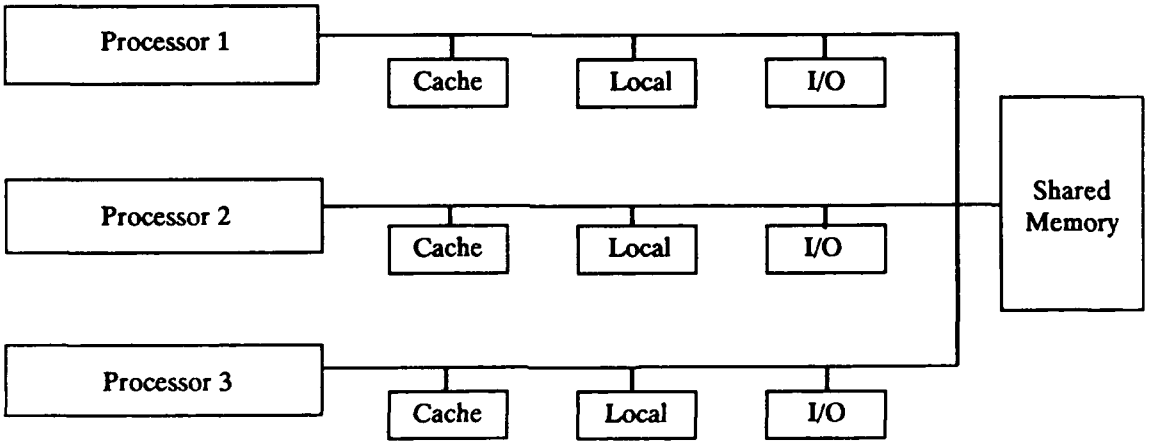


Figure 3-3
Shared Bus System

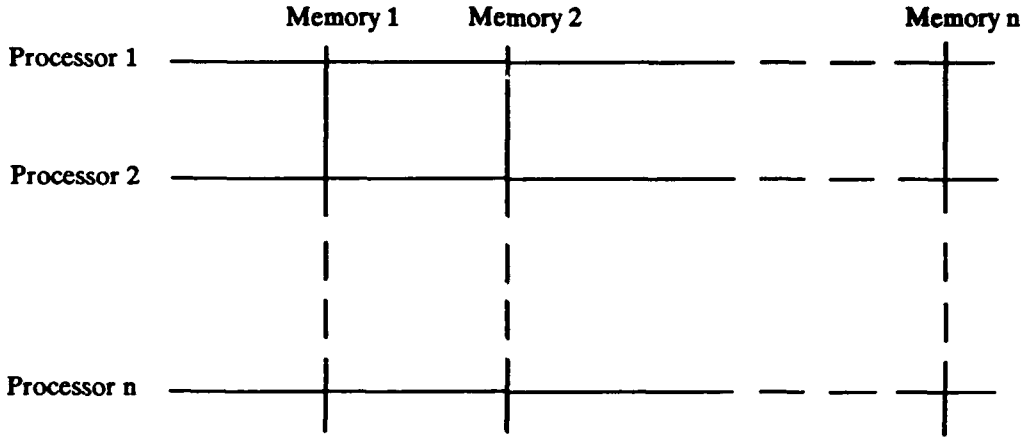


Figure 3-4
Crossbar System

Crossbar

The second type of communication network is the crossbar switch. Stone [88] refers to the switch as the "anthesis of the bus" because it offers the least contention, but has the highest complexity. Figure 3-4 illustrates the crossbar switch. In the figure are N processors connected via the crossbar to N memories. Usually the number of memories is equal to or a small multiple of the number of processors.

In this type of system, a number of processors can have simultaneous access to the memory modules as long as there is no conflict. For example, processor 1 can be connected to memory 2 while processors 2 and 3 are connected to memories 1 and 4 respectively. Thus, the crossbar switch has the

capability of allowing up to N simultaneous accesses provided that no two processors access the same memory. In cases where contention does occur, a contention algorithm decides which processor will gain access to the memory first, putting the other one on a queue.

The complexity of the crossbar switch has limited its use in commercial applications. One example of this communication system was Carnegie-Mellon's C.mmp system which was comprised of 16 DEC PDP-11 minicomputers connected to 16 memory modules by a 16 x 16 crossbar switching system.

Hypercubes

The third type of communication network utilizes hypercubes. Because hypercubes allow for a large number of processors, this type of toppology is popular in fine grain systems. Figure 3-5 illustrates the hypercube topology. Associated with each figure is the dimensionality of the hypercube. A single node is a zeroth order hypercube, a line a first order cube, a square a second order cube and a cube a third order hypercube. To make an nth order hypercube, all one needs to do is take two copies of the n-1 order hypercube and connect corresponding nodes together. For example, to build a 4th order hypercube, take two third order cubes and connect the corresponding corners of the cubes together.

The name, hypercube, is derived from the way the nodes are each interconected to a small number of immediate neighboring nodes. One big attrac-

tion of this type of topology is that the number of connections grows relatively slowly with the size of the hypercube. Specifically, for a nth order hypercube, there are d = 2n nodes with n = log2d connections to each node. A second feature is that the nth order hypercube has all the subsets of connections of the lower order hypercubes.

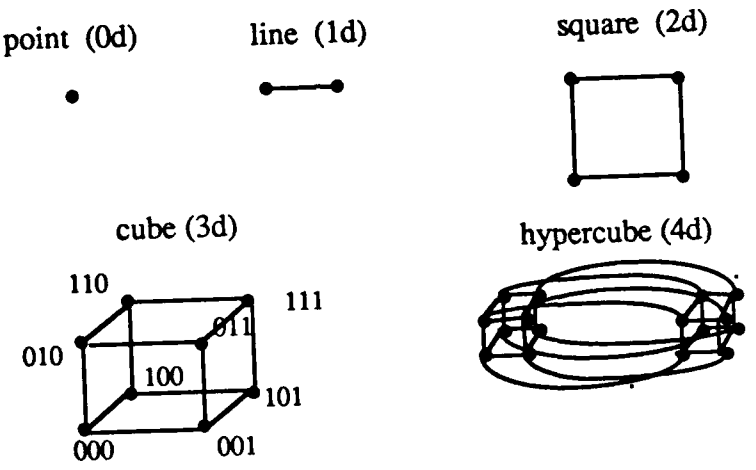


Figure 3-5
Hypercube Architectures

Although not fully connected, the hypercube topology provides a means of communications with remote processing units. If a message needs to be sent to a remote node, a simple algorithm is used for routing the message through intermediate nodes until it reaches the intended node. In an N dimensional hypercube, the log2N nodes connected to a particular node only differ from their

neighbors by exactly one bit position in a binary tag. For example, in figure 3-5c the 000 node is neighbors with the 010, 001, and 100 nodes, all of which only differ in one bit position. At each stage in the routing scheme, the message is sent to the neighbor whose binary tag agrees with the tag of the ultimate destination in the next bit position (e.g. when going from left to right) that differ between the sender and the final destination. For example, in sending a message from 000 to 111 it would first go to 100, since the left bits differ, then to 110, since the middle bits differ, and then finally to 111.

From above, we can see that the path length for sending a message between any two nodes is the number of bit positions in which the binary tags differ. Thus, the maximum path length is simply the dimensionality of the hypercube. Also, note that other paths do exsist and that these redundant paths can be exploited to increase the fault tolerance and enhance the communication bandwith of the system [Heath 85].

Examples of multiprocessor systems utilizing hypercube topologies include INTEL iPSC, Cosmic Cube, NCUBE Hypercube, JPL MARKIII, and the Connection Machine.

Multistage Switching

The last type of communication networks to be discussed are the multistage switching networks. Multistage switching networks use two or more levels of intermediate switching nodes to provide links between the multiple

processors and memories. The network uses a small number of these switching nodes and a clever arrangement of internode links to guaranteee at least one path between each processor and memory. These types of switching networks go by many different names depending on their specific function. Some examples are the Benes, Omega, and Banyan networks. Figure 3-6 shows one generic class of multistage switches called shuffle exchange networks.

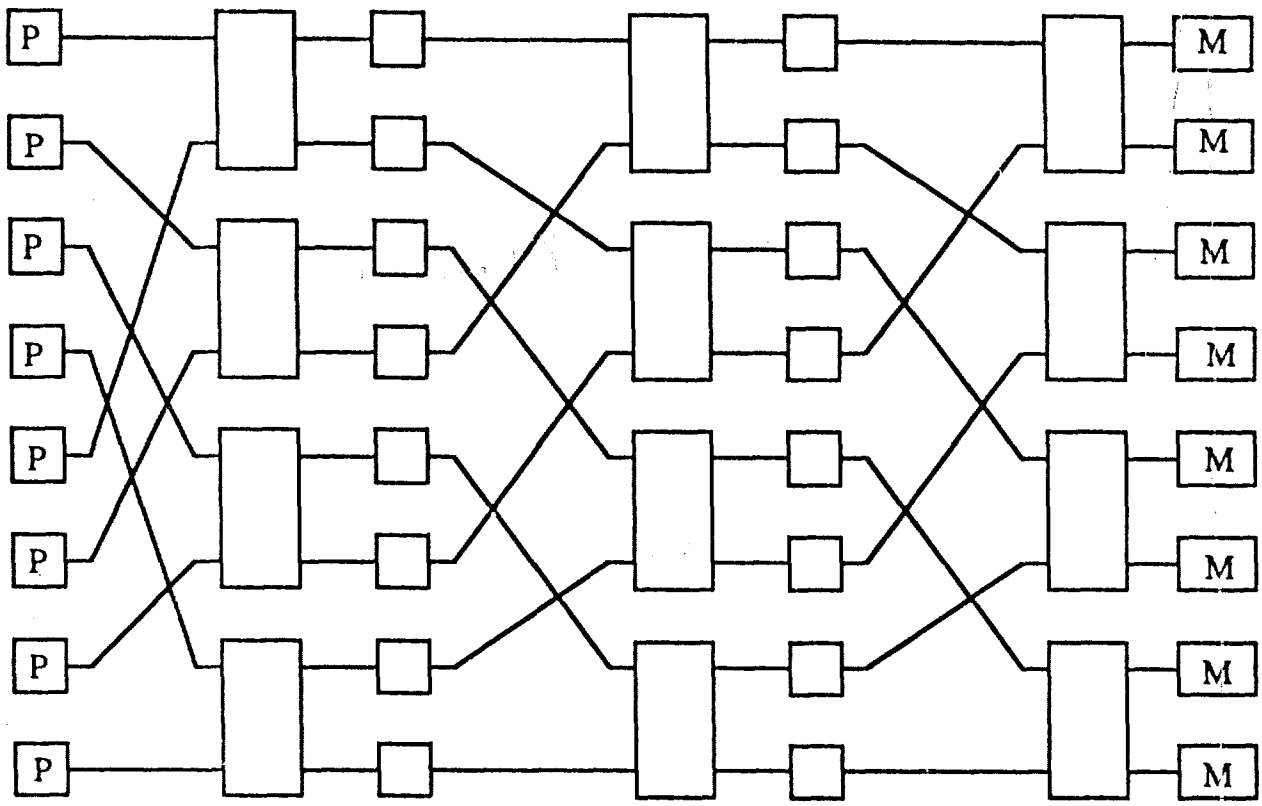


Figure 3-6
Shuffle exchange network with eight processor and eight memories. (From Stone)

In the figure, eight processors are connected to eight memory modules via three levels of switching nodes. This type of network offers an important ad-

ditional feature, the combining switch. This switch is used to reduce contention by performing operations in parallel within the switching network that would otherwise have to be serialized at the memory [Stone 88].

The multistage switching network has certain advantages over the bus and crossbar switching networks. On the bus and crossbar, exclusive access to shared memory is the limiting factor in performance . Once access to a memory is saturated (e.g. 1 for the bus and n for a n x n crossbar) no additional performance can be realized by adding any more processors. However, with the multistage switch, exclusive access to memories can be done in parallel by making use of the combining switches and the memory. Lawrie [75] has shown that if N processors place simultaneous synchronized request so that processor i requests data from memory i + c, for any constant c, the requests can be honored simultaneously without any conflict. Also, no contention occurs if processor i requests data from memory pi + c, where p is an odd number, provided that the number of processors is a power of 2 [Stone 88].

One drawback of multistage switching networks is a problem addressed by Pfister and Norton [85] concerning "hotspots". Hotspots occur when a specific memory module is referenced to the point of contention. The module cannot accept any new data, so the switches connected to the module become backed up since they cannot output data to memory (i.e. a queue of memory accesses is set up). This, in turn, causes a further back up in the preceding switching layer. This can result in the interference of communications to other nodes

in the system that are not related to the hot spot. For example, consider the situation depicted in figure 3-7. Memory 2 is saturated causing the switching modules to be blocked as indicated. Suppose that the demands of processors 2 and 4 caused the contention in memory 2. One result of this is that the path for processor 3 to memory 4 is now blocked, although neither processor 3 or memory 4 were involoved in the saturation.

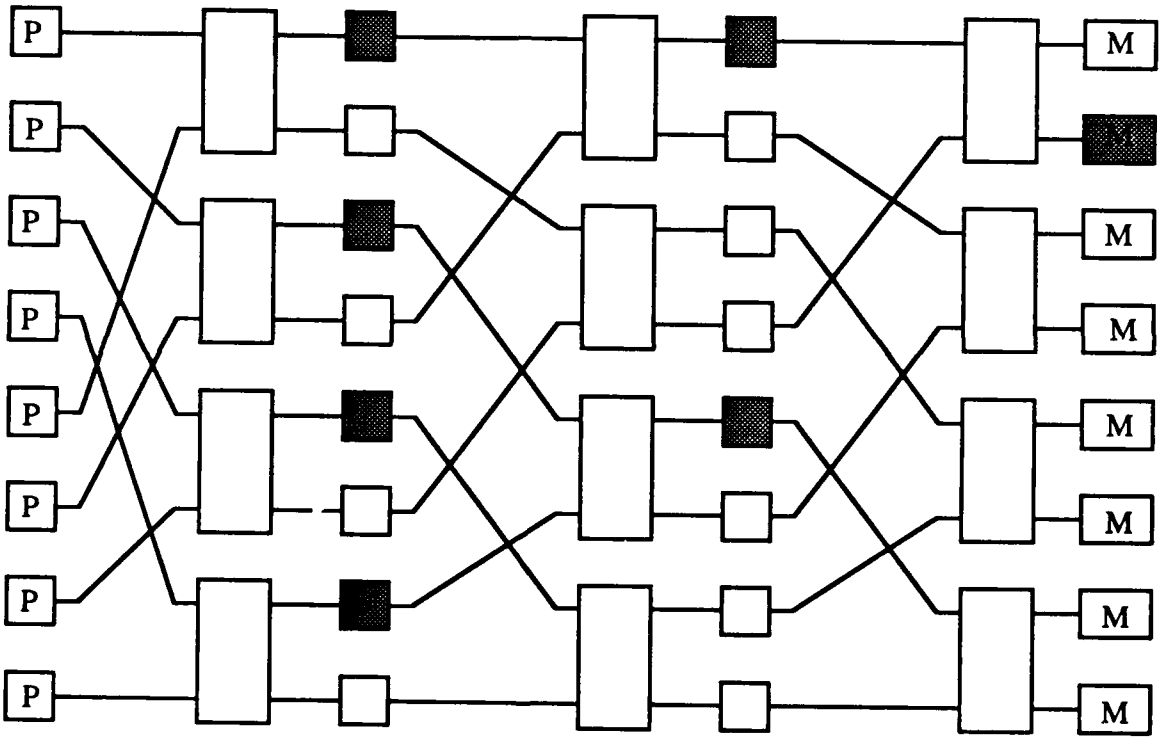


Figure 3-7
Shuffle exchange network with hot spot in memory two. Shading shows blocked switching modules.
(From Stone)

In spite of this drawback, multistage switching is becoming and attractive alternative to bus oriented and crossbar switching systems. Examples of multi-

processor systems using this switching technique include the BBN Butterfly, IBM RP3, and the Illinois Cedar system.

Masscomp 5700 Computer System

As can be seen from the previous sections, multiprocessor architecture is determined by the number and types of processors and memories, and the topology created by the communications network. In this section we will examine the architecture of the Masscomp 5700 multiprocessor system.

The Masscomp 5700 computer is a coarse grain, tightly coupled multiprocessor system. The system is built around the 32-bit Motorola 68020 microprocessor architecture and will support up to eight CPU modules. The communications network of this system is based on a triple bus architecture. The overall system architecture is comprised of the following elements:

- Up to 8 Central Multiprocessing Units (CMPU)
- One or more MULTIBUS Adapters (MBA)
- One or more Central Memory Modules (CMM)
- An Auxilary Function Module (AFM)
- Three Buses; Synchronos Memory Interconnect (SMI), Masscomp enhanced MULTIBUS and STD buses
- MULTIBUS controller modules (e.g disk, graphics, ETHERNET)
- One or more Data Acquistion/Controller Processors (DACP)
- STD + data acquisition devices (e.g. d/a, a/d converters)

Figure 3-8 illustrates a typical multiprocessor configuration. The following paragraphs describe the major system components and how they interact.

Buses

One of the key elements to the Masscomp system is its triple bus architecture. Each bus is a collection of interface signals that allows the systems's hardware elements to interact with one another. This interaction includes I/O data transfers, direct memory access (DMA) transfers, and the generation of interrupts.

These buses use the master-slave concept of control. That is, a master device initiates bus activity by taking control of the bus and sending the address of the device it wishes to communicate to on the bus. After detecting its address the other device becomes the slave and executes the command sent by the master. Different buses implement the master-slave relationship using various protocols that define how the master and slave are to interact.

The three buses used on the Masscomp system include two industry standard buses and Masscomps own bus. The two commerical buses, the MULTIBUS and STD buses, have some Masscomp enhancements. These particular buses were chosen to assure compatibility between Masscomp's system and peripherial equipment from other manufacturers. The third bus is Masscomp's own Synchronous Memory Interconnect, or SMI bus.

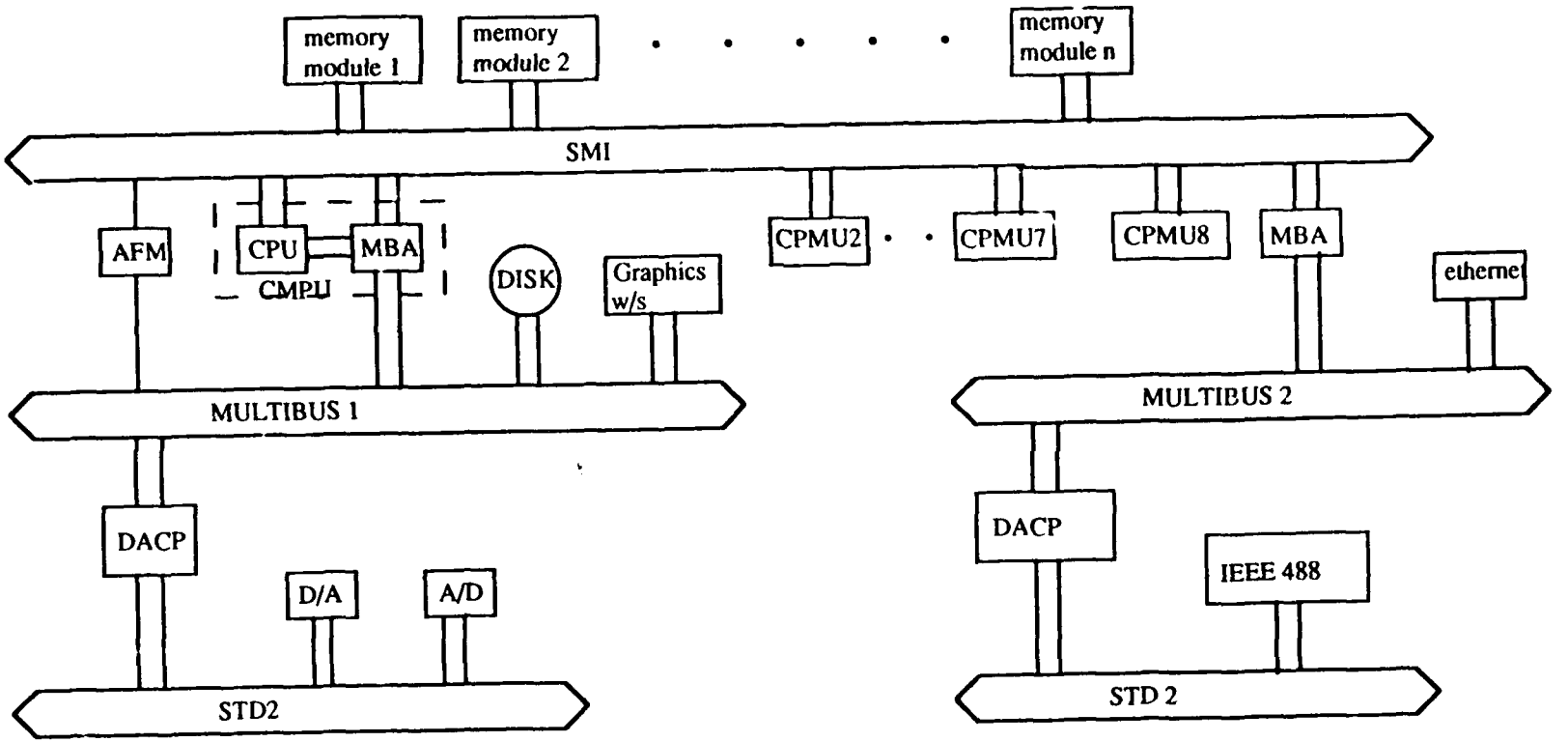


Figure 3-8

Typical Masscomp multiprocessor configuration

The STD+ bus is used exclusively for Masscomp data acquisition devices such as A/D, D/A, and parallel I/O modules. Masscomp calls this the STD+ bus because they combined two 9-slot STD buses side-by-side . While each STD bus share address lines, they have seperate data lines. It utilizes the typical master-slave relationship described above. In this particular system however, the DACP module always acts as the master and the device modules as the slaves.

The second bus in the system is the MULTIBUS. This bus conforms to the industry standard IEEE-796 MULTIBUS. It has several Masscomp enhancements that allow up to a 6M Bytes/sec transfer rate versus the standard 2.5 - 3.5M Bytes/sec. The bus is used by intelligent I/O controllers (e.g. the MBA and DACP) handling graphics, mass storage, and data acquisition.

Masscomp added several extra lines to the standard MULTIBUS to enhance its features. The biggest enhancement is the use of a block mode control signal that allows the time multiplexing of 32 bits of data onto 16 data lines. This extension provides for twice the throughput for Masscomp devices designed for the higher throughput, and, at the same time, maintains compatibility with those standard devices using the IEEE specifications.

The last bus is the Synchronous Memory Interconnect. This bus is used to link all the CPU modules with the main memory modules over a dedicated, high speed path. The bus is designed to support multiprocessing, allowing uniform access by all processors to physical memory.

One of the features that enhances the SMI for effecient multiprocessor use is Masscomps communication algorithm called "split transaction" protocol. This protocol is used to arbitrate between the devices on the bus. Any device on the SMI that requests a transfer of data (e.g. CPU, graphics) sends out a specific command over the bus then releases the bus. At some later time, the device responding to the command (e.g. main memory) gains control of the bus, responding to the request. Thus, at any point in time all devices are eligabel to control the SMI.

The purpose of this "split transaction" is to make more effecient use of the bus. For example, in the more common interlocked protocols, once a device has control of the bus it does not relinquish it until the transaction is complete. This means that bus cycles are wasted while the master device is waiting for the slave to fetch the requested data. With the split transaction, a high bus bandwidth is achieved because that idle bus time is freed up, allowing other devices that are waiting for the bus to make their requests.

These three buses are interconnected through two dedicated Masscomp devices; the MULTIBUS adapter (MBA) and the Data Acquisition and Controller Processor (DACP). The MBA handles all communications between the SMI and the MULTIBUS while the DACP handles all communications between the STD + bus and the MULTIBUS. The devices are intelligent I/O controllers, and as such have the necessary circuitry to accomidate differences in control lines, data sizes, and address translations between their respective buses.

CMPU Modules

The Central Multi-processing Unit (CMPU) module contains the central processing resources for the system. It contains the 68020 microprocessor, memory management hardware, an 8K byte cache, a Motorolla MC68881 floating point coprocessor and floating point accelarator, and the optional MULTIBUS adapter. Only the memory management, cache, and MBA will be discussed because of their direct relevance to parallel processing.

Memory Management Hardware

Each CMPU module contains memory management hardware to control the 4G Byte virtual memory address space (32 bit processor thus 2^{32} addresses) using demand paging. In the demand paging used by Masscomp, 4K Byte chunks of contigous address space, referred to as pages, are copied into main memory only when explicitly referenced by a running process. To accomplish this, two primary hardware components are used, a page table engine and a translation buffer.

When a program is read into physical memory, one page at a time, each page's virtual address is first mapped to the physical address that is actually used. The page table engine provides this address mapping or translation. The table generates entries to page tables which are stored in main memory. The page tables keep track of the physical location of each virtual page, with a unique set of page tables being kept for each process. A high speed RAM in the

CMPU module, called the translation buffer, contains the address translations that have most recently been mapped by the page table engine. Thus, a copy of the most frequently used translations is already known to the CPU and need not be fetched from main memory.

The translation buffer is used by the CPU to check if a virtual address has already been translated. If the address is present, a hit occurs. This means that the virtual address can immediatly be converted to the proper physical address. If the address is not present, a miss occurs. The page tables in main memory are then checked to see if the translation is present. If so, the translation buffer is updated. If not, the Operating System fetches that page in secondary memory, which contains the requested address and the page table engine translates the address, updating the main memory page table and the CPU translation buffer.

MULTIBUS Adapter

As shown in figure 3-8 the CMPU has two communications paths, one to the SMI and one to the MULTIBUS. The reason behind this is that the MBA is an optional feature. If the CMPU needs only to communicate with main memory, a MBA is not required. In the figure, CMPUs 2 - 7 are examples. However, since mass storage devices are required as well as other I/O devices, at least one MBA is required for the system. It should also be noted that a CMPU module can access another module's MBA. Thus, a CMPU

without an MBA still has access to all the system's devices even though it has no MBA of its own.

Cache

The CMPU modules utilize an 8K byte, two-way associative cache to minimize the number of wait states incurred by the microprocessor when accessing main memory. The cache is set up with 8 bytes/block and 2 blocks/set. Because the cache has a load through capability and its block size is 8 bytes, after the first 4 bytes (i.e. 32 bits) are available, the CPU can process the data while the cache is receiving the next 4 bytes.

The cache also utilizes a read-allocate, write through algorithm. If data is available in the cache for a read, then it is simple read into the processor. If a read generates a cache miss, then the processor accesses main memory for the data and the 8 byte block that contains the data is loaded into the cache (read-allocate). In write operations, if the data is in the cache, then both the cache and main memory are updated (write through). If the data is not in the cache during a write operation, then only the main memory is updated.

Since multiple processors can use the SMI and share the same memory resources, data modified by one processor must be invalidated in the caches of other processors. Each processor therefore, has an invalidation stack that contains the addresses currently held in the cache. These addresses are checked against the addresses currently held in the cache. If a match occurs, that data is

invalidated and the next request for that data causes a cache miss, which in turn causes the processor to retrieve the updated data from main memory.

Because the cache is two-way associative, a replacement algorithm is necessary. The Masscomp caches use the least recently used (LRU) replacement algorithm. In this particular architecture, this means that if both blocks of the cache, for a given index, are full, the block that has gone the longest time without being referenced is marked for replacement.

Auxiliary Function Module

In this multiprocessor system, one CMPU module is used as the boot processor for the system. Associated with this one processor module is the auxiliary function module (AFM). This module provides a number of system features that would be redundant if on every CMPU module in the system. While the AFM is not an SMI or MULTIBUS device, it provide general services for the buses and CPU.

The AFM's functions can be broken up into five areas:

Bus Arbitration - Arbitration for both the SMI and MULTIBUS, granting the use of each bus to the device with the highest priority.

Bus Clocks - Generates the clocks for the SMI and MULTIBUS buses.

Termination Network - Provides a network of resistors to attenuate bus signals so that reflected signals are not interpreted as valid responses.

Intialization circuit - Provides circuits to detect system power-up, resets, and power outage.

Serial Interface - Generates the TOD clock for the boot CMPU and provides the serial communications between the boot CPU and NVRAM

Central Memory Modules

The system is capable of holding up to 128M Bytes of physical memory in Central Memory Modules (CMM) that each contain up to 4M Bytes. Each module is organized with a 32-bit wide data path and 7 additional check bits assigned to each 32 bits of data. The check bits enable each module to correct single bit errors and detect all double bit errors.

The modules are interleaved in sets of two. Because the system is organized in quadwords (64 bits), each consecutive longword (32 bit) address is physically assigned an alternate module. This particular type of interleaving scheme was set up for two reasons. The first reason is that it allows the CMMs to match the SMI bandwidth and the 8 byte/block requirement of the cache. A single memory module is capable of only utilizing half the bandwidth of the SMI becuse of data fetch delyas and cycle time. By interleaving two modules, one module can have 4 bytes (i.e. 32 bits) of data being transferred on the bus while the other is fetching another 4 bytes. Thus, maximum use of the SMI bus

is achieved, and if its a cache request, it can fill the block with the minimum amount of delay.

The second reason for two module interleaving is module failure. If the modules were totally interleaved, (i.e. each module contains the next consecutive physical address) the failure of any one module would affect the entire 128M bytes physical address space. By having only pairs of modules interleaved, a failure of one module would only affect a localized portion of memory, 8M bytes in size. Thus, using this type of interleaving, the affected address space in a failure is significantly reduced.

SUMMARY

From the preceding sections, one can now see that multiprocessor systems involve more than just adding more processors to a system. The number of processors and how they are interconnected, the location of memory modules, which processors are allowed to access which memory modules, and the type of communications network utilized, are all interrelated and important considerations in the design of an efficient multiprocessor system.

Chapter IV

Implementing Neural Networks on Multiprocessors

Introduction

According to Bower [88], the efficient implementation of programs on multiprocessors is important in neural network research for two main reasons. First, depending on the number of processors and the systems' overall efficiency, the time spent in network development and evaluation can be significantly reduced. Second, larger, more complex networks can be run in the same amount of time it takes a single processor system to run simpler networks.

However, the implementation of neural networks on multiprocessor systems is not all that easy. Because multiprocessor systems have unique network architectures, they also possess special programming requirements. The basic premise behind parallel processing is that the various processors in the system work in parallel, each on its own assigned task, to solve some overall problem. As such, programs written for multiprocessors must be able to be partitioned; distributing the workload among all the processors. Once distributed, there are other requirements to consider such as, how much communications between

processors is needed, and how much synchronization is necessary. Considerable effort has been expended over the past several years in understanding how to efficiently implement these requirements [Bower 88, Kallstrom 88, Kruatrachue 88, Wolfe 88].

To accommodate these requirements, most systems use library extensions to conventional operating systems and languages, enabling the programmer to specify processors and the actions to be taken. Examples of systems using this method, are the Sequent Balance, Masscomp, Alliant FX-8, Connection Machine and Intel iPSC. Other systems use new program languages designed specifically for multiprocessors. Probably the best known example is the OCCAM language used with Intel transputers.

A third method for handling parallel processing requirements is smart compilers. These compilers search for parallelism in sequential programs, freeing the programmer from the job of parallelizing the code. Unfortunately, these compilers are limited in the amount of parallelization they can detect, and they must be tailored to the specific type of microprocessor used, as well as the overall system's architecture.

Thus, for the most part, it is up to the person writing programs on multiprocessor systems to specify the parallelism. Depending on the particular system, this may mean adding calls to execute a routine on one or more processors, or writing a simple piece of code that specifies communication and data paths, and how to partition the program.

Regardless of the method used, one must address the following four key concepts for the effecient implementation of a program on a multiprocessor system:

- Decomposition
- Load Balancing
- Communication Overhead
- Synchronization

The next four sections detail these key concepts.

Decomposition

Decomposition can be defined as a method of breaking up and distributing a single problem among all available processors. The effecient decomposition of a problem is dependant on both the problem to be solved (neural nets in our case) and, the system architecture. Thus, the first logical step in decomposition is to choose a "problem domain" (as Bower [88] refers to it) to fit the architecture of the system. For example, in a system where the number of processors equals, or exceeds the number of nodes in a neural network, the domain could be the individual nodes. In situations where there are more nodes than processors, the decomposition may take place at the layers. That is, all nodes in a layer would be processed by the same processor. Generalizing these two examples, one might characterize the problem domain in terms of the repetitive, physical attributes of the problem, either at a very fine level (one

node/processor), or at a more coarse level (several nodes/processor). In most cases, this type of decomposition preserves spatial relationships. That is to say, nodes close to one another in the network are "mapped" to processors close to one another on the computer system.

The decomposition need not be based on physical attributes, it could very well be based on the network's functions. For example, in a backpropagation network, one or more processors could be assigned the task of summing the inputs to the nodes, other processors, to calculate the activation levels, and still other processors, to perform the error propagation calculations. Another example would be the counterpropagation networks. Since one layer uses a Kohonen algorithm and the other a Grossberg, the network could be decomposed based on these two different functions. In this type of decomposition there is no space preserving mapping from the network nodes to the processors.

One last type of decomposition is based on the system's interprocessor communications capabilities. Although communications will be discussed later, it is worth mentioning how it is a basis for decomposition. In some networks, communications between nodes is very dense in some places, while in others it is sparse. An example would be in a winner-take-all network, where each input is fanned out to a cluster of richly connected nodes, but between clusters, there are few connections. Depending on the specific architecture of the multiprocessor, the nodal clusters might have to be assigned to the same processor because interprocessor communications is sparse, as in say, a ring type topology. If, in-

stead, each node was assigned a processor in this type of network, the time spent in communications among the processors representing a single cluster might result in excessive execution times.

Regardless of the basis for decomposition, the main goal is to distribute the work load in such a manner that all the processors are efficiently utilized. As will become apparent, there are many factors that influence the optimum decomposition of a problem.

Load Balancing

One factor to consider in the decomposition of a neural network is that of load balancing. Load balancing is defined as the process of balancing the processors' work load to keep them uniformly busy. Thus, in the decomposition of a problem, it is not enough to assign tasks to processors, the workload associated with each task must also be considered. For example, in the back-propagation network mentioned above, processors assigned the task of summing inputs have no where near the computational load of those assigned the error propagation task. Thus, to balance the load, more processors would be assigned to the error propagation task than to the summing task.

Balancing processor loads can be accomplished in two ways; statically and dynamically. In static load balancing, the programmer or compiler decides on how to decompose the program before it is executed and no decomposition changes are allowed once the program begins execution. Dynamic load balanc-

ing, on the other hand. allows for adjustments in decomposition during program execution to compensate for dynamic changes in computational loads. With this type of load balancing, extra code is required to monitor these changes and redistribute the loads. Thus, a significant amount of overhead is incurred with dynamic load balancing. Still, in some cases, it results in increased performance, thus it is an option to be considered [Bower 88].

Regardless of which method is used, there is general agreement that optimal load balancing is achieved when the problem can be decomposed into as many concurrent modules or "grains" as possible [Kruatrachue 88]. These grains can be "packed" onto processors in such a fashion as to balance the load, making the most effective use of each processor.

Making the most effective use of each processor does not however, guarantee an optimum execution time. Load balancing by itself often yields decreased performance due to unavoidable communication delays. In fact, there are cases where the execution time on several processors was greater than that of one processor for applications with intensive communications [Kruatrachue 87], [Wolfe 88]. In these cases, although load balancing was effectively used, it was the lack of regard for the associated communications overhead that caused the excess execution times. Thus, another very important factor to consider in decomposition is communication overhead.

Communication Overhead

In the context of parallel processing, communication overhead is that communication necessary when one processor needs information from another processor to perform its task. This overhead shows up as part of the workload associated with each processor. That is, the computational load of each processor can be broken up into two parts. One part is that work associated with performing the actual calculation, and is present in both single and multiprocessor systems. The second part is that work associated with interprocessor communications, and is therefore unique to multiprocessor systems.

In general, communication overhead is costly in terms of time and resources. For example, in hypercube architectures with each processor representing a node, communication overhead is high if the interaction between nodes that are not neighbors is intense. Messages must be passed via intermediate processors to the final destination processor, which takes time and require extra resources (i.e., the intermediate processors). Thus, decompositions considered along the spatial mappings lines describe above, have to take into account the communication overhead costs.

Unfortunately, in trying to minimize communication overhead, load balancing usually suffers. There is a "push-pull" relationship between communication overhead and load balancing. In the extreme case of zero communication overhead, a single processor is assigned all the tasks, which results in the ultimate load imbalance. While the inverse is not necessarily true, a per-

fect load balance does require a less than minimum communication overhead. Thus, an optimal decomposition must strike a balance between these two opposing factors. A lot of research is being done to try and find a general method for determining the optimal decomposition Wolfe 87, 88]. Unfortunately, decompositions are so problem dependant that a general method does not seem possible.

Synchronization

The final factor that must be addressed in the efficient implementation of a program on a multiprocessor system, is synchronization. Once a decomposition is chosen that achieves load balancing and minimizes communication overhead, *the individual processors must somehow be synchronized.* Synchronization is important because excessive execution times could result if one processor is forced to wait for the results of another before it can continue its task. Or even worse, invalid results could occur if the processors were not properly synched.

How the processors are synchronized, depends on the particular decomposition. For example, if each processor represents a single neural network node, the processors could be synced such that they all execute the same instruction at the same time. This type of synchronization is referred to as complete, and is a technique often used in SIMD type computer systems.

However, other decompositions are such that different processors are performing different tasks at different times. Depending on the exact decomposition and amount of interprocessor communications required, asynchronous and loose synchronization would be required. In cases where the load balancing is such that a task is completed by one processor before a second processor needs that information, no synchronization is really necessary. Thus, the processors can actually work asynchronously.

While this is the ideal type of synchronization, that is, no synchronization at all, most decompositions would probably require a fair amount of interprocessor communications. In these cases, what is referred to as loose synchronization is required. In this type of synchronization, the processors are allowed to operate asynchronously *until* such time that one or more processors require information from another processor before they can continue their assigned tasks. If the other processor does not have the required information at that time, the other processor must then wait until it becomes available. Thus, loose synchronization is a cross between being completely synchronous and completely asynchronous; it allows a great deal of flexibility in syncing only those processors that need to communicate. As will be discussed in chapter V, loose synchronization require very little overhead in terms of extra program code to synchronize the various processors at various times.

Multiprocessor Performance Metrics

To analyze the effects of communications overhead and load balancing for various program decompositions, a set of metrics must be developed. In doing so, one can find the optimum decomposition and maximal speedup for a given program. Additionally, a metric must also be developed to compare the performance of multiprocessor systems to that of a single processor system. With this in mind, the following set of metrics were developed based on Stone's [88] R/C ratio concepts for determining how much overhead (C) is incurred per unit of computation (R), and similar work by Bertsckas and Tsitiklis [89].

Three basic assumptions were made in developing these metrics:

- That an average task time can be used which is close to the calculation time incurred (Tcalc) by any task on any processor.
- When processors need to communicate with each other, an overhead of Tcomm(i) is incurred. Also no communication overhead is incurred if tasks are being executed on the same processor and the tasks need to communicate.
- Some portion of communication overhead operations lengthen the total processing time because the overhead cannot be fully overlapped with calculations.

Given these assumptions, one can think of the execution of a task on each processor as consisting of two phases. The first phase is the calculation phase, in which the processor is performing the calculations required to com-

plete the assigned task. The second phase is the communication phase, in which information is being exchanged with other processors.

With this in mind, we can now define average calculation and communication overhead times.

$$T_{acalc} = \frac{1}{N}\sum T_{calc}(i) \qquad 4.1$$

$$T_{acomm} = \frac{1}{N}\sum T_{comm}(i) \qquad 4.2$$

Where $T_{calc}(i)$ is the time required for processor i to make the calculation for some assigned task, and $T_{comm}(i)$ is the communication overhead time required to relay that information to another processor.

Using these definitions, what is the total execution time for an N processor system with M total tasks? To derive the general formulas, first consider the two processor case. With two processors, the total execution time can be approximated by:

$$T_{tot} = T_{acalc}\,Max(M-k,k) + T_{acomm}(M-k,k)$$

where there are M total tasks and k are assigned to one processor and M-k to the other processor. In equation 4.3, the first term denotes the calculation time for two processors. Since the processors operate concurrently, the calculation time is for the larger of the run times for both processors. That is, the larger of $T_{acalc}(M-k)$ time for one processor and $T_{acalc}(k)$ for the second processor. The second term denotes the pair wise communications that take place. For ex-

ample, if there are six total tasks and one processor is assigned two tasks, then, in a worst case scenario, four of the tasks on one processor would need to communicate with the two tasks on the second processor, for a total of (6-2)2 or 8 T_{acomm} units of time.

Extending from two to N processors, we would have k_i tasks assigned to the ith processor and equation 4.3 becomes:

$$T_{tot} = T_{calc}\, Max(k_i) + \frac{T_{acomm}}{2} \sum_i k_i(M - ki)(i)$$

$$= T_{acalc}\, Max(k_i) + \frac{T_{comm}}{2} \sum_i (M^2 - k_i^2) \qquad 4.4$$

In this equation, the first term denotes the largest calculation time incurred among the N processors with k_i tasks. The second term is again the communication overhead, where the *pair* wise combinations of k_i task and $M\text{-}k_i$ tasks are summed. The 1/2 term comes into play because the summation does not take into account for how the pairs are ordered, thus combinations of pairs are counted twice in the summation. The M^2 term results because the summation of ki over N processors is simple M, therefore we get M*M or M^2.

To approximate the sequential execution time for the same program, only the calculation time has to be considered. Thus, we only need to sum up each processor's calc times:

$$T_{seq} \approx \sum_i T_{calc}\,(i) = N * T_{acalc} \qquad 4.5$$

Using these equations, we can now define speedup, efficiency, load imbalance, and communication overhead.

$$\text{Speedup} = \frac{T_{seq}}{T_{tot}} \qquad 4.6$$

$$\text{Efficiency} = \frac{T_{seq}}{(N * T_{tot})} \qquad 4.7$$

$$\text{Load Imbalance} = \frac{T_{acalc}\ \text{Max}(k_i)}{T_{acalc}} - 1 \qquad 4.8$$

$$\text{Comm Overhead} = \frac{T_{acomm}}{2T_{tot}} (M^2 - \sum_i k_i^2) \qquad 4.9$$

These equations now form a set of metrics that can be used to measure multiprocessor performance for various decompositions. They also give an approximation of how to assign tasks to processors, and when parallel implementation is cost effective, as will be explained below.

Stone[88], Indurkhye[86], and Nicol[86] have shown that, depending on the ratio of calculation time to communication overhead time, minimum execution time can be realized by one of two task-to-processor assignments. They found that if:

$$\frac{T_{acalc}}{T_{acomm}} \geq \frac{M}{2} \qquad 4.10$$

then an even distribution of tasks among all N processors produces the fastest execution time. In this context, even distribution means that if M is a multiple of N, then each processor is assigned M/N tasks. If not, then most of the

processors are assigned the integer value of M/N with another processor getting the remainder. For example, with 22 tasks for eight processors, seven of the processors would be assigned three tasks and the eighth processor, one task. It should also be noted that there is the possibility of some processors not being utilized in an even distribution. For example, 22 tasks with seven processors would result in five processors being assigned four tasks, the sixth processor assigned two tasks, and the seventh processor not being assigned any tasks.

In the event that:

$$\frac{T_{acalc}}{T_{acomm}} < \frac{M}{2} \qquad 4.11$$

then no matter how many processors are available, no task assignment produces a faster execution time then assigning all the tasks to a single processor.

Several observations are worth noting here concerning the measure of task calculation time to communication overhead time. First, this ratio highlights the "push-pull" relationship mentioned earlier concerning communication overhead and load balancing. The ratio should be large to prevent the communication overhead from becoming excessive, but at the same time it should be small to create a large number of concurrent tasks. A second observation is that computational efficiency does not necessarily guarantee a significant multiprocessor speedup. For example, the larger the ratio, the more efficient the program computation because of the relatively shorter period of time spent for

communication overhead. However, if the large ratio is the result of decomposing the problem into several large concurrent tasks, as opposed to many tasks, the amount of parallelism is small and thus speedup is limited [Stone 88]. This implies that it is not enough to rely on the calculation/communication ratio; the other metrics must also be examined to determine overall performance.

Summary

The implementation of a program for multiprocessor execution is not an easy task. One must consider how the program can be decomposed, how to achieve an efficient load balance, the effect of communication overhead, and how the tasks between processors need to be synchronized. The number of processors, the particular multiprocessor architecture (especially interprocessor communications), and how they are related to the characteristics of the specific program must also be considered in the implementation. As such, the development of a set of metrics to measure load imbalance, speedup, communication overhead, and efficiency is essential for the efficient decomposition of a program.

Chapter V

Multiprocessor Methodologies

Introduction

Based on the four key concepts addressed in chapter four, I developed several methodologies for the implementation of neural networks on multiprocessor systems. With one exception, the methodologies were made as generic as possible, without regard for a particular type of neural network or multiprocessor system. In doing this, the methodologies would have the widest possible applications.

These methodologies serve as a "blueprint" for network decompositions that can be applied to a wide range of multiprocessor and neural network architectures. That is, it details how the network is to be decomposed, how the load and communications are to be balanced, and how to synchronize the various tasks. However, just as a blueprint, it does not specify what "tools" to use to accomplish these tasks because different systems utilize different tools. For example, the blueprint may specify that the results of some task on proces-

sor A be communicated to all other processors. Depending on the particular multiprocessor, this may mean passing messages via other processors to communicate the information, or setting up a common, shared block of memory on another system. Thus, having the blueprint specify what is to be done and not the specific manner, enables the methodologies to be machine-independent and portable.

A total of four methodologies were developed. Two were based on general network topology and are called the Layer and Cross-layer methodologies. The other two were based on the functional aspects of the network, irrespective of network topology and are called the Pipeline and Hybrid epoch-pattern methodologies. The next four sections present each methodology in detail and the rationale behind their development.

It should be noted that in these methodologies, it was assumed that for any practical networks the number of nodes in the network would far exceed the number of processors. Otherwise, another logical decomposition would have been the assignment of one processor per node.

Layer Method

In the Layer approach, the network is decomposed on the basis of how the nodes are layered in the network. Figure 5-1 depicts how processors are assigned to each layer. As shown, one processor is assigned to each layer. If

more processors are available, then the layer can be subdivided as indicated by the dashed lines.

With this type of decomposition the processors would be kept busy by pipelining data into the network. That is, after the first processor (layer 0) processes information for data set P, its' results are passed on to processor two (layer 1). Then, the first processor begins processing data from data set P + 1. Thus, depending on the number of layers, the pipeline is full after N data sets, where N is the number of layers in the network. Figure 5-2 illustrates the pipeline for a two layer network (remember the input layer is usually not counted in the numbering of layers).

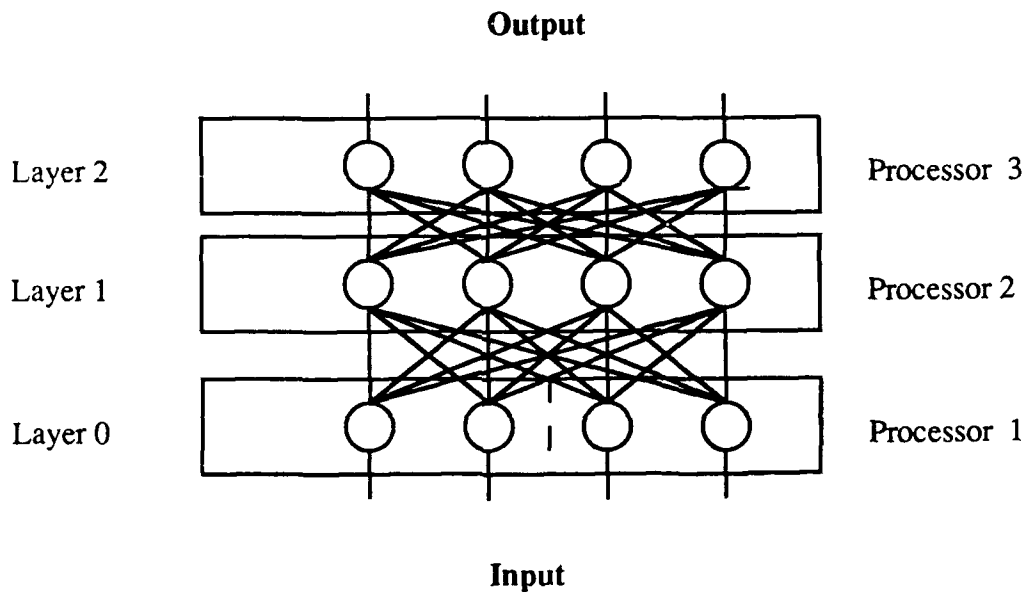


Figure 5-1
Layer Method of Network Decomposition

Depending on the particular algorithm being implemented and the number of training patterns, the concurrency realized by the layered pipeline is variable. For example, in figure 5-2 assume there are a total of six patterns used for training the network and that the weights are adjusted after the sixth pattern. Further assume, as a worst case scenario, that the weights must be modified on a layer by layer basis (as in the case for backpropagation). This then means that each processor must operate sequential when adjusting the weights as shown in the figure. Thus, full concurrency of all the processors is only realized for four time periods out of the total of twelve it takes to process six patterns and adjust the weights. If one were to train over twenty patterns, then out of a total of 26 time periods (2 for fill time + 2 for "draining" + 4 weight adjust + 18 full pipeline) eighteen units would be spent in full concurrency. Thus, in the first example, the best possible speed up would be less than 2 (22/12) with an efficiency of approximately 60 percent (23/(3*12)) and the second, a speed up of approximately 2.5 (64/26)with an efficiency of approximately 80 percent (64/78). Generalizing for P patterns with three processors the speed up is :

$$(3*P+4)/(P+6)$$

Thus, as P becomes much greater than 6, the speed up approaches a maximum value of 3 with 100 percent efficiency.

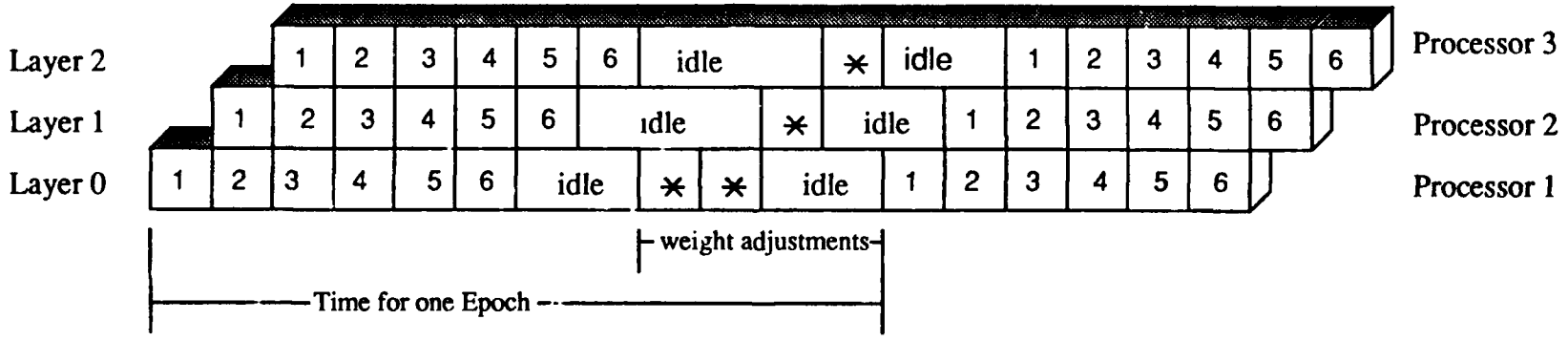


Figure 5-2

Pipeling of data sets through layers

The numbers in each block indicates which of the six data sets per epoch the processors are working on. Because weight adjustments (indicated by *) must be done sequentially, they are not overlapped on the processors.

It should be noted that the time units are relative to the number of processors per layer. For example, if there is one processor per layer it is not unreasonable to approximate the time for the execution of a task in the pipeline as being the same (discounting any communication overhead) for a single processor executing the tasks sequentially. However, with more than one processor per layer, the time to complete a task is significantly reduced. And, since in most networks, there is no intra-layer communication, no communication overhead is incurred by assigning more processors to the same layer. Thus, the speed up for that layer would be close to ideal (Other types of overhead are incurred, such as the time it takes to execute the extra code for implementing a second processor, that prevent ideal speed ups from being achieved).

Load balancing for this methodology is straight forward. Each processor is assigned the number of nodes in each layer. If there is a big discrepancy in the number of nodes from one layer to another, additional processors could then be used in the layer with the larger number of nodes. For example, if layer zero contains ten nodes; layer one; twenty, and layer two; ten, then layer one would have two processors assigned to it.

The communication overhead aspects of this method are more multiprocessor architecture dependant then the other factors. In a bus oriented system, overhead could be kept down by having the results of each layer stored in shared memory locations, preferably with memory interleaving. The processors

would have to be synced such that as one processor was accessing memory for a new data set, the other processors would be processing their respective data.

In a hypercube architecture, to minimize overhead, neighboring processing units would represent different layers of the network. This concept is illustrated in figure 5-3. The figure shows two processors dedicated to each layer. Since most networks do not require intra-layer communications between nodes, the processing units representing those nodes do not have to be adjacent to one another. However, to minimize communication times, the processing units representing adjacent layers should be as physically close to one another as possible, as shown in the figure.

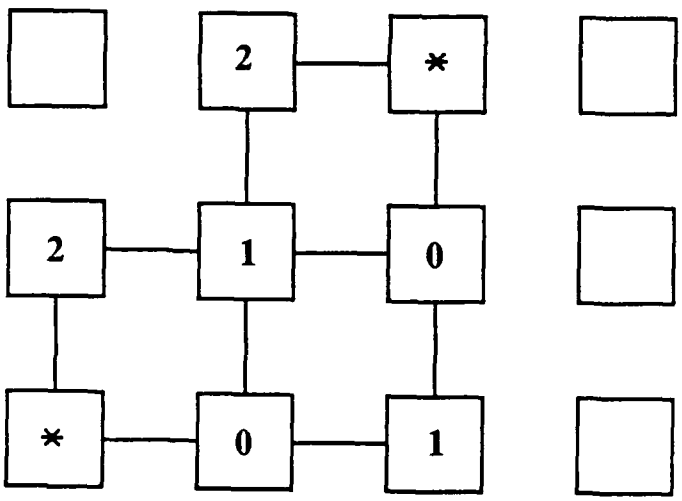


Figure 5-3
Layer assignment for a 2d Hypercube architecture

The number indicate which layer is assigned to which processing unit (PU). Note that most PU's directly connect with their next respective layers. The PU's marked * can be used as alternate communication paths for the layer 1 PUs to communicate with the layer 2 PUs.

As alluded to earlier, synchronization between processors representing different layers is also very important in this methodology. Each processor can operate asynchronously until it reaches a critical section of code when it requires information from another processor. Since the tasks are evenly distributed, execution times for each task should be approximately the same. Offsetting the time each processor starts its tasks would enable the processors to remain in loose synchronization with one another. However, to minimize this offset time and account for variations in processor execution times, semaphores should be utilized. The semaphores will enable the offsets to be optimized and thereby minimizing the communication overhead. They will also act as resynchronizers in the event that one or more processors are delayed in outputting their results.

Cross-Layer Method

A variation of the Layer method is the Cross-layer method. In this method, instead of assigning the processors on the basis of nodes within a layer, processors are assigned to nodes across the layer. Figure 5-4 illustrates this method's principle of decomposition. From the figure one can see how this methodology assigns processors to vertical columns of nodes as opposed to the Layer's method of assigning horizontal layers.

Also, unlike the Layer method, the Cross-layer method does not use pipelining of the data sets. Because of the vertical decomposition, it would be

a logistical nightmare in trying to keep the data sets in proper sequence and be able to properly adjust the weights. Instead, this method maintains a tighter control over the processors almost to the point of operating as a SIMD device. Exactly how this is done will be explained below.

The main purpose of this methodology is to eliminate as much inter-processor communication as possible. To see how this is possible, again consider figure 5-4. In each vertical column there are four inputs (two from layer one nodes and two from layer two nodes) and four outputs (two from layer zero nodes and two from layer one nodes). Had the network been decomposed by layers there would have been nine inputs to layer one from layer zero and nine outputs from layer one to layer two. Thus, with the Cross-layer decomposition, there is a significant reduction in the amount of information that must be shared between processors.

This method is also advantageous in that large networks can be easily decomposed in a balanced manner. The majority of the neural networks being simulated today are not very high (i.e. have a small number of layers) but are fairly wide (20-100 nodes per layer). Thus, with a vertical decomposition dividing the network up into nodes segments of equal size is a fairly trivial problem. Even the assignment of nodes to an odd number of processors is easily handled, as was shown above. The same cannot be said for layer-based decompositions.

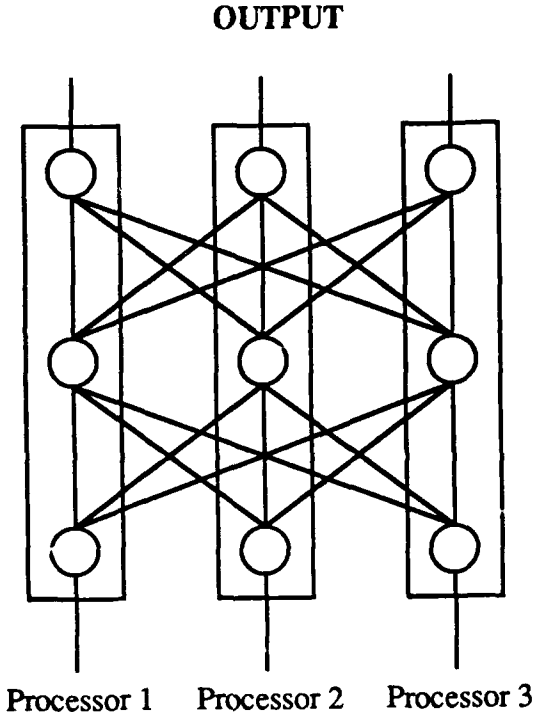


Figure 5-4
Cross Layer Decomposition

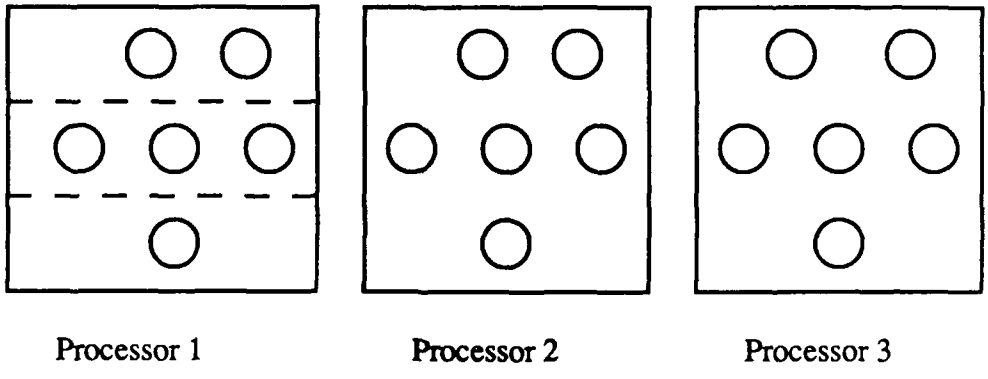


Figure 5-5

Cross Layer Decomposition with differing number of Node per layer (Connections omitted for clarity)

Load balancing is accomplished in a manner similar to the one described in the Layer method. The only difference being that vertical slices of the network are taken as opposed to horizontal. To illustrate how load balancing can be achieved, figure 5-5 shows an example where the number of nodes in each layer is different. If additional processors are available, then further decomposition can take place by dividing each cross layer section into horizontal slices as indicated by the dashed lines.

Communications overhead with this methodology is a somewhat easier task for several reasons. One reason, as pointed out above, is that there are fewer inter-processor lines of communication to worry about. Secondly, because information flows up and out in most networks, the upward flow of information is within the processor. This means less overhead to worry about in terms of coding and communications.

Unfortunately there is a tradeoff between the communications overhead and the synchronization with the Cross-layer method. In simplifying the communications overhead with a vertical decomposition, the synchronization of the processors is more critical. A tighter synchronization is required because each processor must assure that the information from each node in each layer is communicated to the other processors at the appropriate time.

To better illustrates this point consider the following. Each layer zero node on each processor can execute its' required computations independently of the other layer zero nodes. However, each layer one node on each proces-

sor requires information from layer zero nodes before they can begin executing their computations. Since all but a few of the layer zero nodes are on N-1 different processors, the layer zero node's results must be communicated to the Nth processor before the Nth processor's layer one node can begin its computation. This is also true for the other layer one nodes on the other N-1 processors. The overall result is the use of more synchronization code then with the layer method. With the Cross-layer method, synchronization coding is required for all nodes with the exception of those in the input layer (i.e. layer zero).

Depending on the particular multiprocessor architecture, the extra execution time incurred by the synchronization code and time delays caused by waiting for information could negate any processor speedup. To minimize these delays, hypercube and bus-oriented topologies need to incorporate the same techniques described in the Layer method. Namely, putting processors representing communicating nodes as physically close together as possible, and placing information in interleaved, shared memory locations.

Pipeline Method

The Pipeline approach examines the functional aspects of the network to determine what tasks can be done concurrently. That is, rather than looking at the network's topology, we examine the network's paradigms to identify any parallelism that can be exploited. This approach is more in line with research being done in the Computer Science field to use several processors on a single

program. However, it is different in that we are assuming the entire multiprocessor is dedicated to implementing a neural network and that no multiprogramming with multiprocessors will take place (i.e. use multiple processors on a single job but have multiple jobs running on the system as well).

There are three easily identifiable levels of parallelism that can be generalized for neural networks:

- Parallelism at the subroutine level, where multiple subroutine calls can be executed in parallel.
- Parallelism at the loop level, where the multiple iterations of a DO loop can be executed in parallel.
- Parallelism at the block level, where the operations of sections, or blocks of code can be executed in parallel.

The data dependance relationship between the subroutines, loops, and blocks is the determining factor in selecting which level of parallelism to exploit for a particular neural network. To better understand this data dependance relationship, consider the following examples.

In a DO loop there could be instances where the data in one iteration of the loop is required by the next iteration. For example, in:

```
do i = 1,n
Var(i) = Var(i-1) * Delta(i) + Thres(i)
endofdo
```

the element of Var computed during iteration i is used during the next iteration i + 1. Thus, in trying to implement this type of do loop in parallel, each processor would have to wait for the other processor to finish before it could execute its' iteration of the loop. Because this type of data dependency is not uncommon in neural network algorithms, this level of parallelism is not desirable.

At the subroutine level, a similar data dependance could exist. One could have the situation where the results of one subroutine may be required before a second subroutine can begin execution. In this situation, it might be possible to offset the start times of the subroutines such that by the time the second subroutine begins the necessary data is available. Another possibility would be to group the data dependant subroutine together and execute them on the same processor. This is exactly what is done at the block level of parallelism.

At the block level, the data dependance can be structured such that each processor operates with as much autonomy as possible. Blocks of code identified as exhibiting parallelism can be checked for data dependance, and, depending on the amount of dependance, could be assigned to the same processor. Because of this flexibility, one has more latitude in making processor assignments with this level of parallelism than with the other two levels. For this reason, and the fact that network paradigms are extremely diverse, block level parallelism was selected as the basis for my network decompositions.

Unlike the previous two methodologies, decompositions using the Pipeline methodology are more labor intensive. This is mainly due to the fact that in order to decompose the network, one needs a good understanding of the particular network paradigm, and as such, the decompositions are more "custom tailored". This does not however, mean that a step-by-step approach to decompositions is not possible; only that it requires more work.

The first step in the decomposition is to determine those sections of code that are suitable for concurrent execution. Depending on the particular multiprocessor this may simply mean running an Operating system routine such as profile in UNIX, or writing several programs to determine the number of times a function is called and the variables required to execute the function.

Once these sections have been identified, the next step is to place them into N blocks, where N equals the number of available processors. To determine how to group these sections into blocks, the amount of computation per section (i.e. load) and the data dependance between the sections should be the governing factors. Depending on the computational load of the sections, one block may have to consist of two or three sections of code whereas another block might only contain one section in order to balance the overall workload. The other consideration will be how much communication overhead will be incurred as a result of the data dependance between sections on different processors.

Because of the strong data dependance exhibited in neural network paradigms, "pipelining" the blocks on the different processors is the most efficient means of program execution. What is meant by pipelining in this sense is that the execution of the various blocks are staggered in time just enough that as one processor completes its first computation, the result is "piped" over to the second processor so it can begin execution.

For example, assume one processor has some code where a DO loop is computing a result which is required by another section of code on a second processor. As soon as the first iteration of the DO loop is completed, the result can be "piped" over to the second processor, allowing the other section of code to begin execution. To take into account for differences in the computation times between the sections of code, the amount of stagger could be varied so as to produce a buffer. That is, wait X iterations of the DO loop in processor one before beginning the execution of the code on processor two.

The piping of the results between processors takes care of some of the synchronization requirements for this methodology. However, to assure that synchronization is maintained, semaphores should also be utilized. If the processors are properly staggered however, the semaphores should not result in any "wait states" unless an unexpected interrupt occurs; throwing off the synchronization.

The other synchronization requirement of the system is for the resynchronization of the blocks. If there is a low data dependency between

blocks on different processors, the processors can operate asynchronously most of the time. However, as new data sets are presented, the processors may need to be resynched to assure they are operating on the same data sets. To accomplish this, a master sync needs to be established. Here again, using a semaphore will suffice in resynching the processors.

Figure 5-6 illustrates the overall process. In the figure there are four processors. Three of the processors are assigned two sections of code each, with the last processor being assigned three sections. Load balancing is such that all four processors have approximately the same amount of work. However, the data dependance is such that processors one and two have a strong data dependency as do processors three and four. As such, the start times for processors two and four must be slightly staggered to allow processor one and three to "pipe" their results. Note that since there is no data dependance between the first two processors (i.e. processors one and two) and the last two processors (i.e processors three and four), these two sets of processors can operate completely independently of one another (save the resynching that must take place). This can be seen by the fact that processors one and three begin execution at the same time.

Also note that there are various periods of time when several of the processors are idle. This is due to the resynchronization that must take place between the data sets. In this particular case, nothing can be done to shorten the idle time because the sections of code on processor three are the limiting fac-

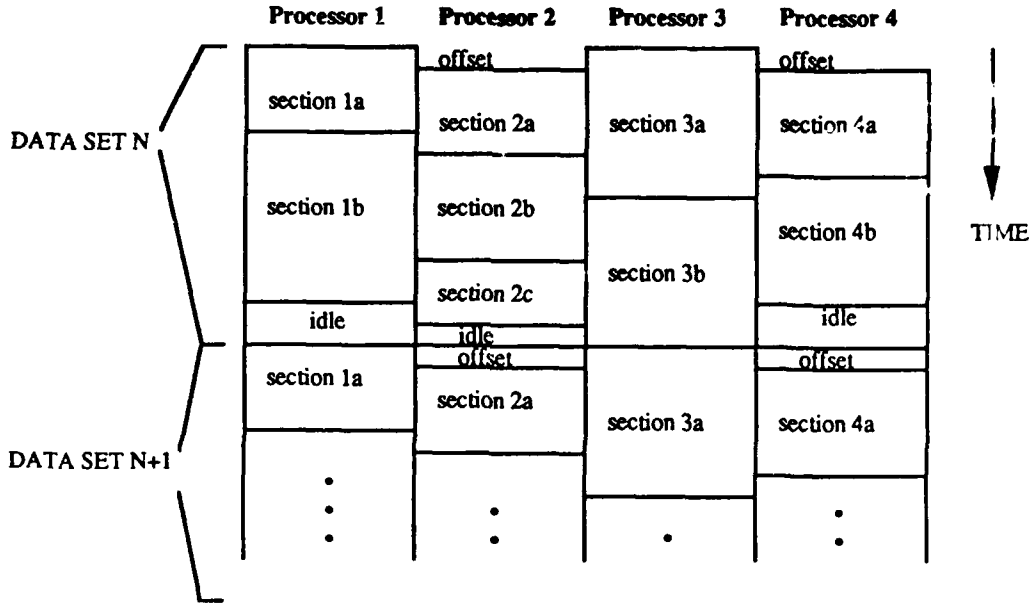


Figure 5-6
Pipeline Approach

tor. That is, there is no idle time on processor three between data sets, therefore the other processors are forced to wait for processor three to complete its' tasks before starting another data set.

Hybrid Epoch-Pattern Method

The Hybrid Epoch-Pattern methodology was developed specifically for error correcting algorithms, such as backpropagation. This particular methodology was developed for several reasons. The first reason was to see how much more speedup might be gained by developing a methodology for a specific class of neural networks. Since backpropagation is by far the most popular network paradigm, an error correcting methodology was chosen. The second reason was

that I have some definite ideas on how to modify the classic pattern and epoch training methods used in error correction networks to make the network converge to a solution in less time on multiple processors.

In classic training methods one either trains over patterns or epochs. In pattern training, weight adjustments are made after each pattern is presented, whereas in epoch training, pattern errors are summed and weight adjustments are only made after all the patterns are presented. In pattern training, if there are a large number of patterns, adjusting the weights after each pattern can be very time consuming. One might therefore assume, for a large number of patterns, that epoch training would be more desirable. However, epoch training has a disadvantage in that weight adjustments would only be made after a considerable amount of computation time was spent doing pattern calculations. Thus, for a large number of patterns, a disproportionate amount of time is spent on pattern computations as compared to the time spent for weight adjustments. The overall result is an exceedingly long time for the network to converge to a solution. In fact, Rumelhart and McClelland [88] state that for large sets of patterns, pattern training is the preferred method, even with its drawbacks.

In the hybrid epoch-pattern method I have developed, I break the patterns up into "pseudo epochs" so as to make weight adjustments after a reasonable number of patterns. For example, if there are 75 patterns for a complete data set, they could be broken up into three "pseudo epochs" of 25. Thus,

after 25 patterns are presented, a weight adjustment would take place. This represents a savings of 73 weight adjustment calculations over the pattern method and a gain of two weight adjustments over the epoch method. I believe that a multiprocessor implementation of this hybrid method would significantly reduce the network's time to convergence.

In using this hybrid method, an anomaly results in how the patterns are grouped for weight adjustments. An example of this anomaly is shown in figure 5-7. In the figure, there are ten patterns that are being trained on a system with three processors. Thus, the epoch size is three. As shown, the patterns are grouped into sets of three, where each set represents the patterns over which the weight adjustments are made. Note that for each pattern there are three different groupings. For example, for pattern one the groupings are (1,2,3), (10,1,2), and (9,10,1). Further note that these grouping repeat, or cycle, after ten pseudo-epoch iterations.

The number of pseudo-epoch groupings between cycles can be generalized to:

P, if N is not a multiple of P

P/2, if P and N are even, but N is not a multiple of P

P/N, if N is a multiple of P

where P is the total number of patterns and N is the number of processors

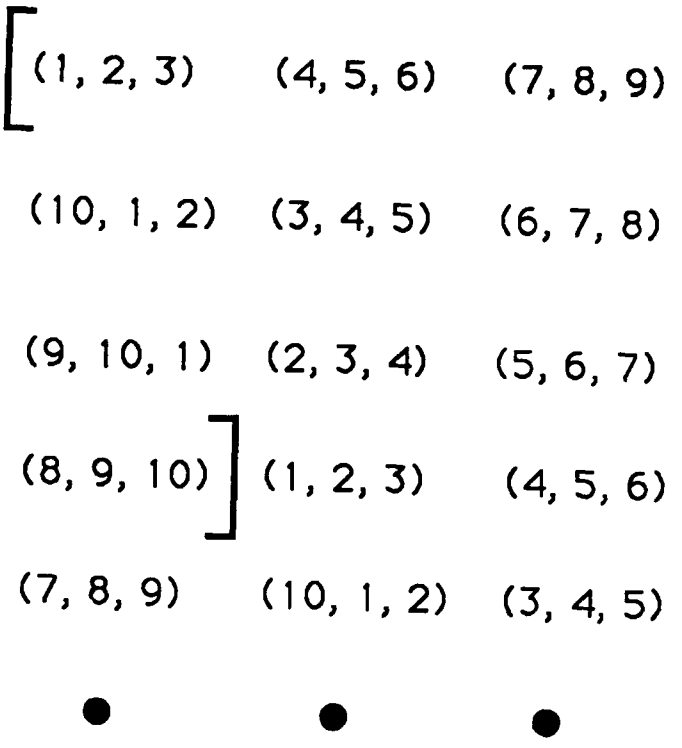


Figure 5-7
Hybrid Epoch Pattern training

The numbers represent the data set for a network and the parenthesis, the "pseudo epochs". Weight adjustments occur after the last pattern in each pseudo epoch. The figure shows the groupings for 10 patterns on a three processor system. Note how the groups cycle (denoted by the brackets) after 10 such groupings.

Thus, although the groupings change after every weight adjustment, they are cyclic. This cycling is important because it provides the researcher with a means to consistently measure intermediate results, and a basis for comparing this training method to the pattern and epoch methods.

Decomposition of networks using this methodology can be accomplished in several ways, depending on the number and type of processors. For a system

with a small number of relatively powerful processors, each processor could be assigned one data set. In this case, the size of the pseudo epochs would be equal to the number of processors. Figure 5-8 illustrates this concept for four processors. Note, that in this particular situation, we are assuming a worst case scenario in that the weight adjustment code cannot be split up between processors, and therefore idle time is incurred by the other three processors. Essentially all that is required in this type of decomposition are multiple copies of the program for each processor, with one of the processors handling the weight adjustments. The processors can operate asynchronously in this case with only minor semaphore coding to allow for the weight adjustment and idle time.

If more processors are available, a second method of decomposing the network would be to use the pipeline method described earlier and group the processors into sets. Each set of processors would then operate on one data set. Thus, in this situation, the size of the pseudo epochs would be equal to the number of sets of processors. Figure 5-9 illustrates this concept.

In this figure, the network functions are decomposed into two blocks, as described in the pipeline section, thus at least two processors are required to implement the network. Since four processors are available, two processors as assigned the task of computing data set i while the other two compute data set $i+1$. The pseudo epoch size in this case is two, meaning weight adjustments are required after every two patterns. As shown in the figure, the first set of

processors (i.e. processors 1 and 2) handle the weight adjustment while the second set of processors are idle.

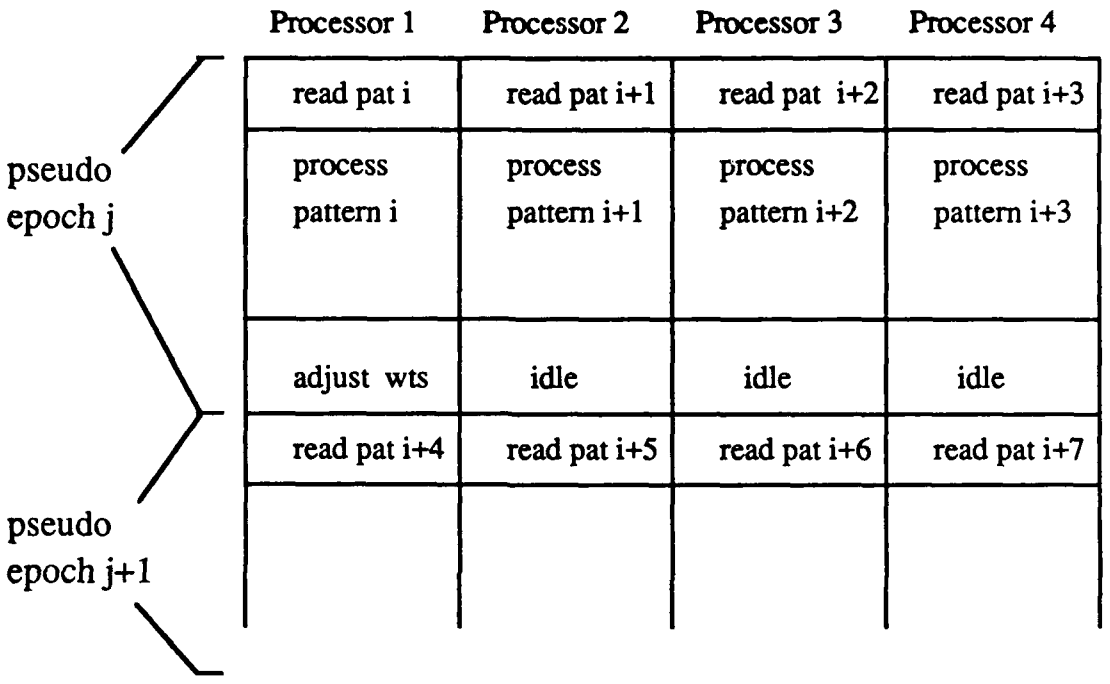


Figure 5-8
Hybrid Epoch Pattern Methodology

The figure illustrates the timing diagram for four processors and the pseudo epochs they create.

In using groups of processors to compute each data set, two levels of synchronization would be required. The first level would be the synchronization of the processors within each group. This is exactly the synchronization that was described in the last section for the pipeline method. The second level of synchronization would be the synchronization required between the groups of processors. With this method, each group of processors would be operating

asynchronously until it came time to adjust the weights. At that point, they would either share the weight adjustment computations, if possible, or one processor would act as a master, while the others were idle.

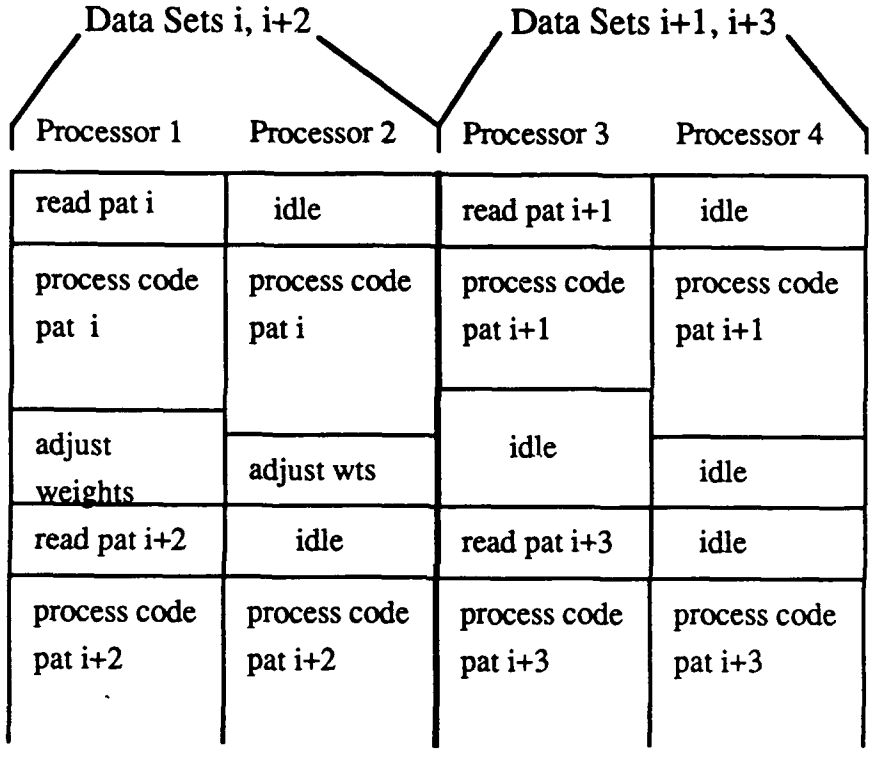


Figure 5-9
Pipelining of pseudo epochs

The figure depicts two processors being used per data set to process the net-

Another important feature of this methodology is that it allows the method of training to be varied anywhere between the classical pattern and epoch methods. That is, the processors can be set up such that the weights are adjusted after every pattern (i.e. pattern method), after the entire set of pat-

terns (i.e. epoch method), or after any number of patterns in between these two extremes.

Example Implementation of Methodologies

As a means of summarizing the information presented in the previous sections, Table 5-1 provides a comparison of the four methodologies. It compares each method based on the four key concepts present in Chapter IV. In addition, it lists the advantages and disadvantages of each methodology. With these four methodologies now presented, it is worthwhile to examine an example network for a better understanding of how these methodologies might be implemented. For the purposes of this example, I will make the following assumptions:

- The network, named the 1881 network, consists of 3 layers with a total of 36 nodes and 224 connections. The network implements the backpropagation algorithm. Figure 5-10 illustrates the network topology.
- The network is to be implemented on a four processor computer system utilizing a shared memory bus.
- The network is to be trained on a 50 pattern training set.

The next four sections detail how the methodologies are implemented.

	Layer	Cross-Layer	Pipeline	Hybrid
Basis for Decomposition	Network Topology within layer	Network Topology accross Layer	Parallelism exhibitedby network algorithms	Parallelism of pattern within epoch
Load Balancing	relatively easy. Each processor assigned one or more layers	Very easy, Each processor assigned vertical slices of network across layers	Moderately easy depending on amount of parallelism exhibited by learning and activation algorithms	Relatively easy. Each processor computes different data set in parallel
Communication Overhead	Highest of all methods. Function of the # of connections between each node in each layer	Moderately high. Function of # of connections across nodes on different processors. Less connections than layer method.	Moderately low. Dependant on number of shared variables	Lowest of all methods. But still dependant on number of shared variables
Synchroni-zation	Realatively easy. But dependant on type of network architecture (bus, hypercube, etc). Information piped to next processor for each pattern	Very difficult. Processor execution needs to be staggered to prevent contention for memory resources	Moderately difficult dependant on amount of parallelism within algorithms	Relatively easy. Asynchronous operation until wgt adjustments required for pseudo epochs
Advantages	Easy to implement	Easy to implement decomposition and load balancing between processors	Functionally based, more latitude in determining concurrent operation	Same as Pipeline. Also exploits strong points of pattern and epoch training
Disadvantages	Communication overhead worst limitation. Overhead grows as size of network increases. Also effeciency limited by # of training patterns	Synchronization worst limitation. Extra coding may limit amount of processor speedup	Speedup limited by amount of parallelism that can be found in algorithm	Number of processors effeciently utilized limited to # of patterns in pseudo epoch

Table 5-1

Comparisons of Methodologies

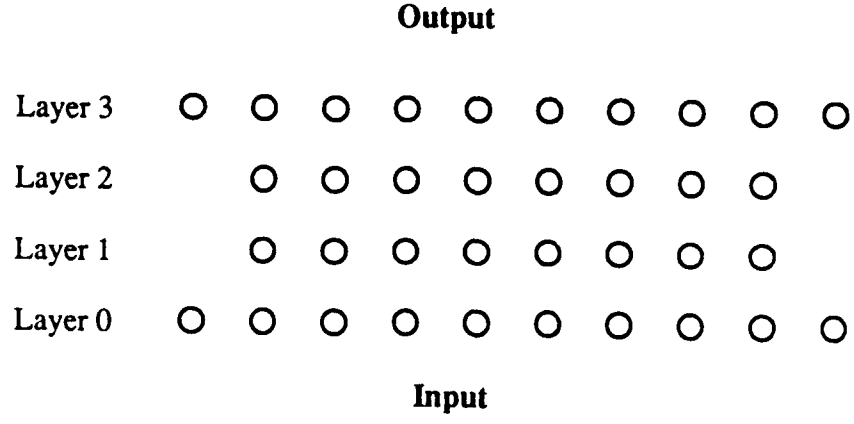


Figure 5-10
1881 Three Layer Network

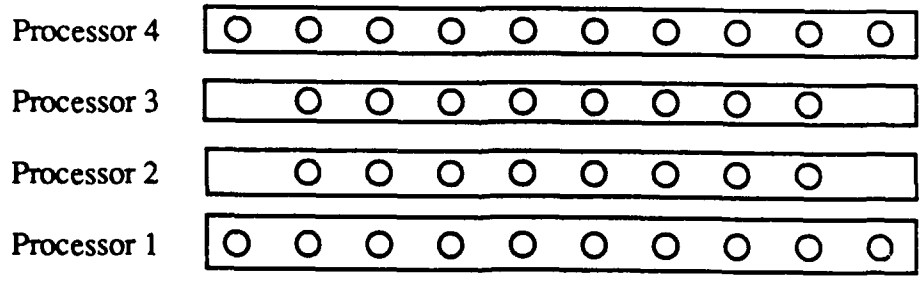


Figure 5-11
Layer Decomposition of 1881 Network

Layer

Figure 5-11 illustrates the processor assignment for the Layer method. As shown, each processor is assigned a single layer of the network. While the connections between each node are not shown for clarity, the number of connections between each processor is as follows:

Processor 1 and 2 80

Processor 2 and 3 64

Processor 3 and 4 80

With this particular network's decomposition, a slight load imbalance exists. Since layer O acts as a distribution point for the inputs, processor one has a much lighter processing load than the other three processors. Also in terms of communications, process two only has 64 activation values to pass along whereas processors one and three each have 80. And, although processor four only has ten outputs to send to the "outside world", it has the extra task of processing the output error (i.e., t_{pj}-o_{pj}) that the other processes do not. Thus, there is a slight imbalance in the load between each processor. Had the number of nodes in each layer been the same, there would have been a better load balance for processor two, but processor one's load would have remained the same.

As alluded to above, the communication overhead for each processor is dependent on the number of connections between each processor. For example, processor's one and three have more information to pass than do proces-

sor's two and four. In terms of the overall communication overhead, when, in time, each processor is actively sending information to another processor, determines the total amount of communication overhead. For example, consider those points in time when all four processors are operating concurrently (figure 5-12, from time step 4 on). Assuming processor one has just started computing data set four, then at times i and i+1, each processor will have pipelined the following data sets:

	Time i	Time i + 1
Processor 1	data set 4	5
Processor 2	data set 3	4
Processor 3	data set 2	3
Processor 4	data set 1	2

After each processor finishes processing its' data set, it must pass the information on to the next processor so it will have it for the next time step. Assuming all four processors finish at approximately the same time, they will all then try to access the memory bus, and contention will occur. Thus, the communication overhead is greatest at these times.

In this particular case however, the load imbalance offsets some of the contention for the memory bus, helping to lower the communication overhead. Because processor one's load is "light", it will finish before the other processors so it will access the bus first. Processor two and three have fewer nodes to

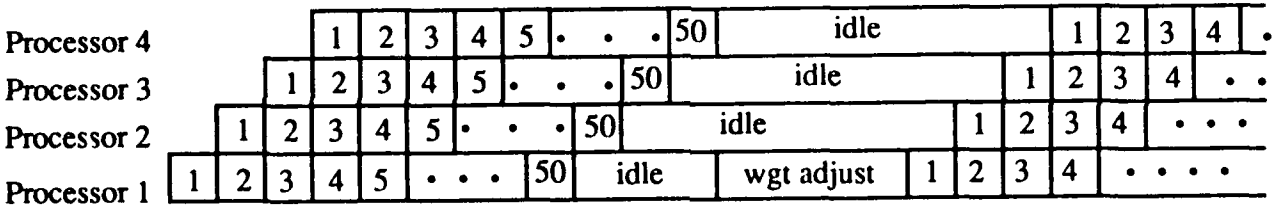


Figure 5-12

Pipeline Synchronization of 1881 Network

process, thus they will finish before processor four. Although there will be contention for the bus, at this point, at least processor two will have less information to pass (64 vs 80).

It should be noted here that the fact that contention occurs is not necessarily bad. In this particular example contention would probably not significantly slow down the overall processing speed because of the small number of connections between the processors. However, as the network size grows to thousands, and tens of thousands of connections between processor, contention would significantly reduce any speedups.

Synchronization of this network is fairly straightforward. Since each processor is working on a different data set and the results are place into memory, the only time synchronization is required is when each processor initially begins its' training set and when processor one performs weight adjustments. Figure 5-12 illustrates the pipelining and weight adjustment for the network. Processor two cannot begin until processor one finishes its results on data set one. A similar situation exists for processors three and four. Once the execution of the training set (i.e. all fifty patterns) is completed, processor one is required to wait for processor four to finish before performing the weight adjustment. As mentioned previously, a simple set of semaphores is all that is required to keep the processors in this loose form of synchronization.

The advantage of the layer method lies in its simplicity; one processor is assigned to each layer, the data sets are "piped", and loose synchronization is all

that is required. The disadvantages of the method however, are the load imbalances and probable communication overhead that will result as the network grows in size.

Cross-Layer

Figure 5-13 illustrates the processor assignment for the Cross-layer method. As shown, processors one and four have two more nodes than processors two and three. The breakdown of the number of connections between each processor is as follows:

	Input	Output
Processor 1	44	44
Processor 2	40	40
Processor 3	40	40
Processor 4	44	44

Note that while each processor has been assigned the same number of nodes as in the layer method, the number of connections in more evenly distributed among the processors with the Cross-layer method.

In terms of load balancing, the Cross-layer method is better balanced then the Layer method. Because the Cross- layer utilizes vertical slices, each processor has part of the "light" load of layer 0, where its only function is to distribute weighted inputs. Each processor in the Cross-layer method also handles

part of the output layer's initial error correction, again a fairly "light loaded" task. Thus, in terms of load balancing, the Cross-layer method is simpler, yet superior than the Layer method.

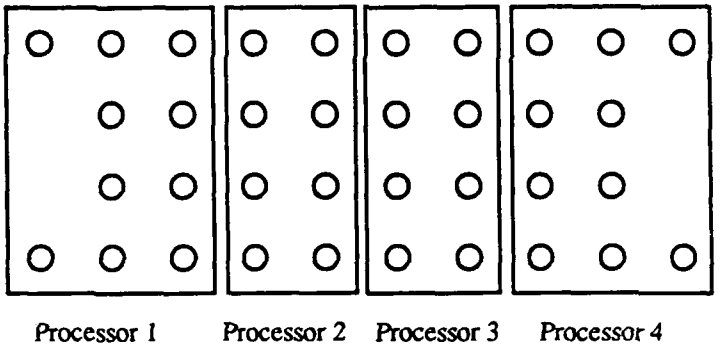


Figure 5-13
Cross-Layer Decomposition of 1881 Network

Cross-layer communication overhead is also somewhat less than the overhead exhibited in the layer method for two reason. First, as mentioned above, the connections are more evenly distributed among the processors. Thus, they all have about the same amount of information to pass and receive. Secondly, because the connections are distributed among several layers, the processor only needs to send and receive short "packets" of data. That is, the processors do not need to send or receive all of their data at once because the data is on several layers, and the layers are processed sequentially within each processor.

The disadvantage of the Cross-layer method lies in the complexity of the synchronization that is required. While synchronization in the Layer method is fairly straight forward, it is more complex in the Cross-layer method. Because

portions of each layer are being processed in parallel on all four processors they need to be carefully synchronized to avoid erroneous data from being processed. Thus, each processor must check to make sure it has received the required data from the other processor before processing layers one, two, and three. This careful synchronization requires extra coding not required in the Layer method. It is possible therefore, that the extra coding would offset any advantages gained in the more effecient load balancing and communication overhead.

Pipeline

Unlike the previous two topological based decompositions, the Pipeline method bases its decomposition on any parallelism exhibited by the network's algorithms. Thus, the first step in the decomposition is to determine those sections of code for concurrent operation and identify an common variables that must be shared.

As mentioned in Chapter II the back propagation algorithm can be divided into four basic steps:

- Compute output
- Calculate error for each output node
- Backpropagate error to previous layers
- Calculate weight changes

Within each of these four basic steps, there are some sub-tasks that can be executed in parallel. For example, in the error calculation, the error array must be zeroed, the initial error calculated, the Least Mean Squared (LMS) error calculated, and the delta error, for those nodes connected to the output layer, calculated. The initial error and the LMS calculations can be done in parallel and the delta error calculations can begin as soon as a node's initial error calculation is completed. Thus, there is full concurrency in the error and LMS calculation, and a "pipeline" concurrency in the delta calculation (By "pipeline" concurrency we mean that once a node's error is calculated the error is "piped" to the next processor where its' delta error is then calculated. Thus, as one processor is calculating node j's error, a second processor can calculate node j-1's delta error).

In addition to this parallelism, some of the four basic backpropagation steps can be executed with "pipeline" concurrency. For example, in the back-propagation of the error and the weight adjustment steps, weight adjustments can begin as soon as the error for a particular node is determined. Thus, one can stagger the time each processor begins those two steps, such that as one process computes the error for a node and stats on another node, a second processor can compute the weight adjustments for the first node.

Figure 5-14 illustrates how the backpropagation algorithm can be decomposed and timed among four processors to take advantage of both full and "pipeline" parallelism. Note that while each processor can share in computing

the output, the error computation cannot start until the compute output function is finished, thus processors one, two, and three are idle for a short period of time.

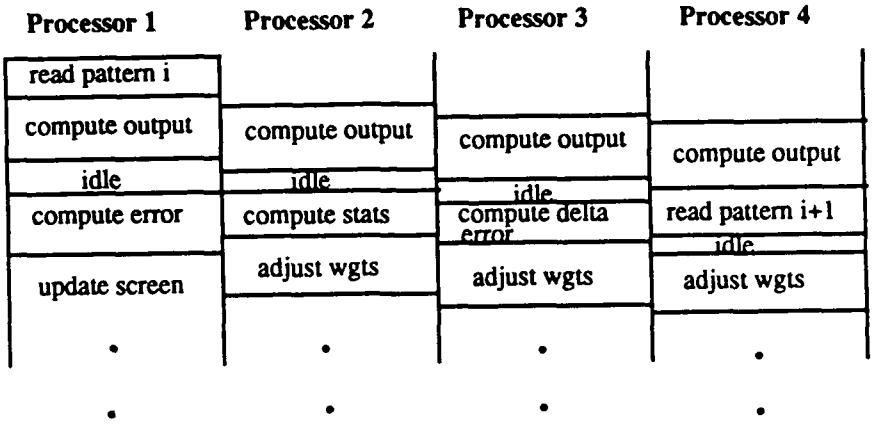


Figure 5-14
Decomposition and timing diagram for 1881 network using Pipeline method

Decomposition and load balancing using the Pipeline method is more difficult than in the Layer and Cross-layer methods because it relies on the amount of parallelism that can be found in the network algorithms. To exploit the parallelism to the fullest, this often requires a thorough understanding of the network algorithms. Thus, the decomposition and load balancing for the Pipeline method are more labor intensive. However, the load balancing adjustment can be made somewhat easier, if a lot of parallelism can be found, for one has more flexibility in processor assignments.

Communication overhead in the Pipeline method is very dependent on the number of shared variables between processors. Unlike the topology based

methods, where the source of the communication overhead is the number of connections, in the Pipeline method, the source is the number of shared variables. From one network to another, the number of variables that need to be shared is different, depending on how the algorithm is written and how much parallelism is found. In this particular example, the processors must share the activation outputs in order for all four processors to compute the output. However, because the two backpropagation function, compute error and compute stats are completely independent, none of their variables need to be shared among the processors. Thus, one advantage of the Pipeline method is that communication overhead is not necessarily fixed, as in the topology based methods. With the flexibility in load balancing, some dependent variables can be assigned to the same processor, thus helping to reduce communication overhead.

The amount of synchronization required of the various processors using this method, depends on the data dependence between each processor. As mentioned above, all four processors have shared variables for compute out, but none for compute error and stats. Thus, the compute output function must be synched among all four processors whereas compute stat and compute error can be asynchronous.

From the above discussion it is apparent that, depending on the amount of parallelism that can be found in the algorithm, the Pipeline method is more flexible in terms of load balancing, communication overhead and synchroniza-

tion. The more parallelism that can be found, the more latitude there is for load balancing. Flexibility in load balancing in turn, enables dependent tasks to be grouped, lowering communication overhead, and making synchronization an easier task. The tradeoff for this extra flexibility however, lies in the amount of time required to thoroughly understand the algorithm in order to exploit as much parallelism as possible.

Hybrid Epoch-Pattern Method

As with the Pipeline method, the basis for the Hybrid Epoch-pattern decomposition is also functional. Figure 5-15 illustrates the decomposition and timing diagram for the 1881 network using the hybrid epoch-pattern approach. This particular decomposition assumes a weight adjustment after eight patterns (i.e., a pseudo epoch of eight). Note that two forms of parallelism are exploited in this example. The first, is simply the processing of a separate pattern on each processor. That is, each processor computes the output and resulting error for a separate pattern in parallel. The second, is the pipeline parallelism exhibited by the weight adjustment function, which makes use of the same decomposition technique explained above for the Pipeline method.

This networks basis for decomposition offers more flexibility than the Pipeline method. With the Hybrid method, one can simply decompose the network by assigning one processor to each pattern, and any one of the processors to do the weight adjustments. Performing a decomposition in this manner is

very easy, although it means the other n-1 processors are idle during the weight adjustment period (as an example see Figure 5-8). For a more rigorous decomposition, instead of assigning one processor to perform the weight adjustments, the parallelism in the weight adjustment function could be examined and exploited, as was done in this particular case. Thus, depending on which route is taken, the decomposition can be very simplistic or involve a study of the algorithms parallelism, as in the Pipeline method.

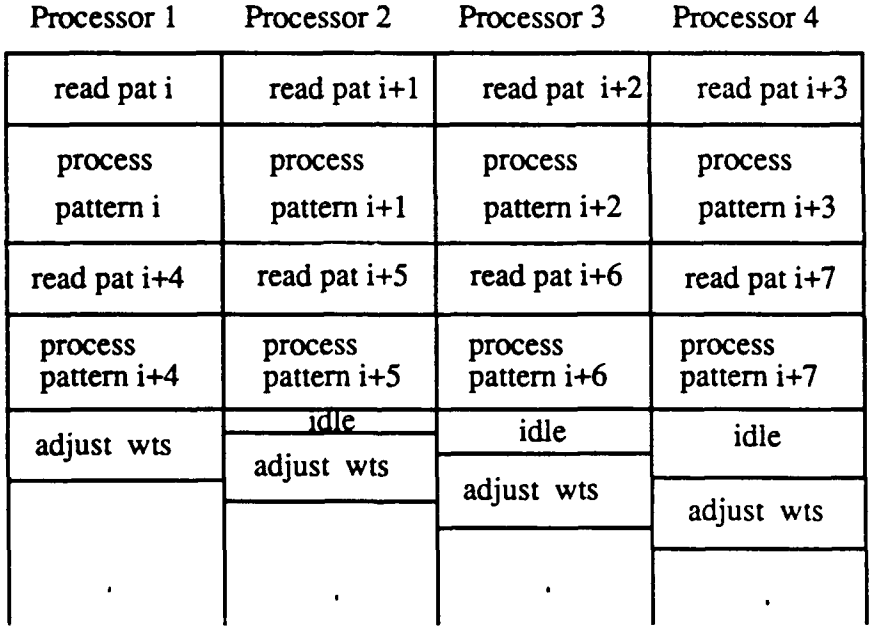


Figure 5-15
Decomposition and timing diagram for 1881 network using Hybrid Epoch Pattern Methodology

Regardless of the decomposition method chosen, the load is very well balanced among processors. Each processor would be operating much like a

SIMD machine where each processor executes the same instructions on a different training pattern. The difference in this case however, is that each processor would operate asynchronously and have its own copy of the instructions.

Communication overhead is low for this method. The only variables that need to be shared between processor are the errors computed for each pattern and the weighted error derivatives (wed) necessary for the weight adjustment function. Thus, the number of variables that need to be communicated between processors is small.

In terms of synchronization, this method allows the processors to operate asynchronously until it is time for a weight adjustment (in this example, after every eight patterns). Even then, synchronization is fairly easy because of the small number of variables that are shared. Unlike the Pipeline method, where there are more shared variables and more functions split between processors, the Hybrid method only has to synchronize the weight adjustment function. Thus, synchronization of the Hybrid method is more easily accomplished than in the Pipeline method.

Table 5-2 summarizes the merits of each method for the implementation of the 1881 network. As shown in the table, each method is compared with the others on the basis of ease of decomposition, ease of load balancing, amount of communication overhead and ease of synchronization. The overall advantages and disadvantages of each method are also summarized in the table.

	Layer	Cross-Layer	Pipeline	Hybrid
Decomposition	Relatively easy	Easy	Difficult, Labor Intensive	Easy
Load Balancing	Relatively easy, but no flexibility	Easy, but no flexibility	Moderately easy and very flexible	Very easy and moderately flexible
Communication Overhead	Highest amount of overhead	Less than Pipeline, but still high	Lower than topology-based methods	Lowest of all methods
Synchroni-zation	Relatively easy	Very difficult	Moderately east	Very easy
Advantages	Easy to implement	Moderately easy to implement	Flexibility in decomposition and load balancing	Easy to implement and flexible
Disadvantages	No flexibility, fixed to topology	Synchronization very difficult	Labor intensive, requires through knowledge of algorithm	Labor intnsive if wgt functions divided amoung processors

Table 5-2

Comparisons of Methodologies for 1881 Network

Extension of Methodologies to other Neural Networks

Based on the principles presented in the previous sections, table 5-3 provides a comparison of the four methodologies for several neural network paradigms. The table compares the ease of implementation and expected performanace of a number of popular neural network paradigms. This table is not meant to be an exhaustive list of neural networks, but rather an indicator of the best methodology for the implementation of several specific networks.

Summary

Four methodologies were developed for the implementation of neural networks on multiprocessor systems. The Layer and Cross-layer methods use the network's topology as the basis for parallel decompositions. While these approaches provide an easy method for exploiting parallelism, the communication overhead is a harder task to solve. The Cross-layer method appears to minimize the communication overhead, but at the expense of extra synchronization code.

The two other methodologies were based on the block level parallelism exhibited by network algorithms. The Pipeline method is generic enough that it can be applied to any network paradigm. The limiting factor in this methodology however, is the amount of parallelism that can be found in any particular network paradigm. As such, network decompositions using this method are more labor intensive then the first two methodologies.

	Layer	Cross-Layer	Pipeline	Hybrid
Adaptiver Resonance Theory	Well Suited Good expected performance	Minimally suited Moderate expected performance	Well suited Moderate expected performance	Minimally suited Least expected performance
Back-propagation	Very well suited Moderate Expected Performance	Very well suited Least expected performance	Minimally suited Good expected performance	Well suited Best expected performance
Cauchy	Very well suited Moderate expected performance	Very well suited Least expected performance	Minimally suited Good expected performance	Well suited Best expected performance
Counter-propagation	Very well suited Good expected performance	Minimally suited Least expected performance	Very well suited Good expected performance	Minimally suited Least expected performance
Boltzman	Very well suited Moderate expected performance	Minimally suited Moderated expected performance	Well suited Moderated expected performance	Minimally suited Least expected performance

Table 5-3

Comparisons of Methodologies for Several Neural Networks

The Hybrid Epoch-Pattern methodology was developed specifically for error-correcting algorithms. It exploits weaknesses found in epoch and pattern training methods when training on large data sets. This methodology assigns one or more processors per data set, enabling the system to execute several data sets concurrently, in an asynchronous manner.

Chapter VI

Applications

Introduction

This chapter examines the application of the four methodologies presented in chapter V to two neural network simulation routines on a coarse grain multiprocessor system. In the sections that follow I will: describe the software and hardware testbed used to examine methodology performance, detail the techniques used to apply these methodologies, and analyze the performance obtained from several network simulations.

Applications Testbed

Table 6-1 lists the hardware and software assembled to test the performance of the methodologies. A Masscomp model 5700 with dual processors was chosen as the multiprocessor system on which to test the methodologies. This UNIX-based, bus oriented system was described in detail in chapter three. This particular system was chosen for three reasons. Firstly, it is representative of most bus oriented multiprocessors with their cache memories, and

EQUIPMENT:	
- HARDWARE	Masscomp 5700 computer system with two 68020 microprocessors (expandable to eight)
- HOST SYSTEM SOFTWARE	ATT Sys V UNIX Operating System Masscomp C compiler
TEST SOFTWARE	
- McClelland & Ruelhart	PDP Backprogpagation Simulator
- Becker	Screen Driven Menu Oriented Neural Network Simulator (developed at UMBA Medical School Div of Medical Informatics)
TEST NETWORKS:	
- 33 different networks ranging from five nodes with six connections to three hundred nodes with twenty thousand connections	

Table 6-1
Methodologies Testbed

local and global memory capabilities. Secondly, its' hardware features were adequate for a proof-of-concept of the methodologies to be tested. And thirdly, I had exclusive access to one such system that was previously purchased with funds from an Air Force grant.

Two neural network software simulators were chosen for implementation on the multiprocessor to test the methodologies. The first simulator chosen was McClelland and Rumelhart's Parallel Distributed Processing (PDP) backpropagation network simulator. It was selected due to its availability. and

the fact that the source code was provided with the simulator. The second simulator chosen was a screen driven, menu oriented (SDMO) neural network simulator developed at the University of Maryland, School of Medicine, Medical Informatics Division. It was selected because of the availability of its' source code, ease of learning, and the fact that it also included a backpropagation paradigm.

Both simulators allow the user to define the following neural network parameters:

- Number of layers
- Number of nodes per layer
- Initial weights and biases
- *Activation threshold*
- Learning rate
- Momentum
- Number of training iterations
- Number of patterns in training set
- Minimum acceptable pattern error

While the screen driven simulator only allows fully connected networks, the PDP version allows the user to specify the connections between nodes.

The fact that both simulators utilize the backpropagation algorithm was not coincidental. I wanted to implement this particular paradigm on a multi-

processor system for several reasons. One reason was that this paradigm is used, in one form or another, in approximately 80% of all ongoing neural network research and development [Jackel 89]. As such, the successful implementation of the backpropagation paradigm with these methodologies might prove of immediate R&D benefit. A second reason was that the availability of two different implementations of the same paradigm would allow me to see if there were any performance differences based on how the same algorithm is written by different people. That is, one backpropagation algorithm might work well on all the methodologies, but a second algorithm, written in a different manner, might not work as well.

Appendix 1 lists the thirty-three neural networks used to test the implementations of the two simulators with the various methodologies. The networks ranged in size from five nodes with six connections to three hundred nodes with twenty thousand connections.

The networks were trained on different training sets depending on the number of input and output nodes in each network. In those networks with an equal number of input and output nodes, two training sets were used. In the first training set, the networks were trained to respond with a right circular shift of the input vector. For example, if the input vector was 01010, the correct response was 00101. In the second training set, the networks were trained to respond with a double right circular shift of the input vector (e.g. 01010 to 10010).

In networks where the number of input and output nodes differed, the networks were trained on random input to output groupings. For example, if the input vector was 01001, the correct output vector was randomly selected to be 0011000100 for a ten node output layer.

The number of patterns in each of the training sets ranged from four patterns up to twenty patterns, depending on the size of the networks. For example, a network with four input and output nodes might have ten different right circular shift patterns in it's training set.

Experimental Design

Both of the simulators were implemented using each of the four developed methodologies, for a total of eight multiprocessor implementations. To compare single and multiprocessor performance, the unaltered, original version of a simulator was first run on a single Masscomp processor followed by a run of the modified version of the simulator (for a single methodology) on both processors. This process was repeated for each simulator until all four methodologies had been implemented.

The single and parallel versions of the simulators both used the same training sets, under the same initial conditions (i.e. weights, biases, etc.). Immediately after the single processor version converged to a solution for a training set, the parallel version was executed for the same training set. This process

was repeated at least three times, with the averaged results used in the analysis of performance.

Because the weights can converge to different values and still result in the same correct answers, the performance data was only considered valid if both the single and parallel versions converged to the same set of weights. In networks with a small number of connections, all the weights were compared. In larger networks however, convergence to the same solution was checked by randomly sampling 500 weights in various layers of both versions, and comparing them with one another.

Modification Criteria

One of the major objectives of this dissertation was to take existing neural network simulators developed for sequential computer systems and implement them on multiprocessor systems. In doing so, I had two major modification criteria that governed how the methodologies would implement the network simulators. The first criteria was that only minimal changes to the existing code would be allowed. My intent was not to rewrite the entire simulation programs, but to adapt them with the least amount of changes. The second criteria was that no modifications were allowed that would change the original "intent" of the network's learning and activation rules. This was of major importance because I did not want the sequential and parallel versions to differ in any way except speed of execution.

Precursors to Implementation

Before implementing the simulators using the four methodologies, I had to develop the code necessary to handle the metrics, decomposition between processors, shared memory, and synchronization. This section briefly describes the various codes.

Metrics

A C program called stats was developed to obtain the necessary metrics for measuring how well each methodology performed. A complete listing of the program is given in Appendix 2. The program enables the user to embed a stats call anywhere in the simulation program to see how well the program is performing. Calls can be placed after every iteration or every epoch to get a detailed analysis of methodology performance. Some of the parameters measured by the program are:

- Elapsed user time
- Elapsed system time
- Shared memory size
- Number of Page faults
- Number of voluntary switches
- Number of involuntary switches

Network Decomposition

To examine the potential parallelism of the simulations for the Pipeline and Hybrid methodologies, the UNIX profile command was utilized. Figure 6-1 shows an example of the output from the profile command. In this particular example, the profile shows the results of 500 iterations of the PDP simulator running an Exclusive OR network. Note that the majority of the time was spent executing the change_weights, compute_error, compute_output and compute_wed functions. These four functions would therefore be the prime candidates for parallel execution.

%time	cumsec	#call	ms/call	name
32.5	5.53	500	11.07	_change_weights
18.2	8.63	500	6.20	_compute_error
17.0	11.53	500	5.80	_compute_output
15.2	14.12	500	5.17	_compute_wed
4.0	14.81	10000	0.07	_logistic
3.2	15.34			_exp
2.1	15.69			_doprnt
0.9	15.84	500	0.30	_settarget
0.8	15.98	500	0.28	_sumstats
0.7	16.10			_display_float
0.5	16.18			_write
0.4	16.25	500	0.13	_setinput

Figure 6-1
Sample Output from Profile Command

To handle the assignment of the concurrent blocks of code to the various processors, several sections of code were written. One section of code

was used to create a child process. A second section would then specify which processor the child process would use to execute its' code. To better understand this process, I will continue with the example started in figure 6-1.

The order of execution for the four functions cited above are:

- compute_output
- compute_error
- compute_wed
- change_weights

Assume that compute_output and compute_wed are to be placed on one processor and compute_error and change_weights are to be put on a second processor. Figure 6-2 illustrates how this can be accomplished. Within the function train(), the function trial() is called. The first time trail is called, it forks, creating a child process. Because the child process has a process id of 0 (i.e.pid $==0$) it executes the if-then clause. In the clause, the system is instructed to use processor two to execute compute_error and change_weights. While the child process is doing this, the parent process continues executing the instructions in train(); compute_output and compute_wed. Through the use of synchronization flags (to be explained below), the child process keeps looping and executing the same section of code until the parent process termi nates the child.

```
train(){
for(t = 0; t nepochs; t+ +){
        read patterns();
        if(!forkflag)
                trial();
        compute_output();
        sumstats();
        compute_wed();
        parentflag = 1;
        if( error  ecrit) break;
} /* end of t */
```

(a) PARENT PROCESS

```
trial(){
        if(!forkflag){
                forkflag = 1;
                pid = fork();
        } /* end of if */
        if(pid = = 0){
                function = mpadvise(MPA CPU SET, 4);
                forkagain:

                compute_error();
                change_weights();

                while(!parentflag)
                parentflag = 0;
                goto forkagain;
        } /* end of pid eq 0 */
} /* end of trial */
```

(b) CHILD PROCESS

Figure 6-2
Sample C code for Parent and Child Process

Synchronization

To handle the synchronization of the various blocks of parallel code, a simple system of flags is utilized. Figure 6-2 illustrates the use of the flags. After compute wed on processor one is finished, it sets the flag, parentflag to 1. This allows processor two to continue, after completing change weights. Note that if processor two does finish before processor one, it simply loops in the while statement until processor one does finish. Afterwards, it resets parentflag to 0 and continues. The purpose of this particular flag was to assure that both processors were operating on the same data set. As such, it was being utilized as a means to resynchronize the processors in the event they fall out of synch.

Shared Memory

To enable the processors to share the variables needed by the parallel blocks of code, and to know the status of each flag, shared memory is required. Figure 6-3 illustrates a section of code where shared memory is defined and assigned. In the code, a section of memory called "lumped" is defined. The two system calls, "shmget" and "shmat", get a specified size of memory allocated for shared memory usage, then, attach the memory to any processors knowing the proper id name of the memory ("lumpid" in this case). After being assigned, the variables that need to be shared between processors can be assigned specific addresses in shared memory. This is shown in the figure for the variable arrays "delta" and "error". It should be noted that the commented out sec-

tions in the figure are the original code that defined and assigned memory to the delta and error arrays.

Layer Method Results and Analysis

Figure 6-4 illustrates a typical decomposition and timing diagram for a four pattern epoch implementation of the Layer method. The networks were decomposed as evenly as possible between the two processors. In cases where there were an odd number of layers, as shown in the figure, processor one was assigned the extra layer. This type of arrangement did not create much of a load imbalance however, because the input layer's only function is to distribute the weighted input vectors to each of the nodes in layer one. This imbalance appears in the figure as an idle period for processor two. What little load imbalance that is present is further reduced in networks where there are a larger number of layers.

As indicated in the figure, the data sets are "piped" from one processor to another. Thus, while processor one is computing data set $i+1$, processor two is computing data set i. Also note, that all the weight adjustments are handled by processor one.

The results of the PDP simulator are shown in figures 6-5 and 6-6. As shown, the Layer methodology never achieved any significant amount of speedup for any of the test networks. Analysis of various execution times indi-

```
/* activation, bias, target, error, delta arrays lumped here in shared memory */

errno = 0; /* create shared memory of proper size */
lumpsize = (unsigned)(sizeof(float)*(4*nunits + noutputs));
/* get the shared memory allocated */
lumpid = shmget(0, lumpsize, 0666|IPC CREAT);

if (errno > 0)
        fprintf(stderr, "errno is %3d, which means %s\n",
        errno, sys_errlist[errno]);
printf("lumpid is %10d\n", lumpid);
errno = 0;

lumppage = calc_page(lumpsize);

/* attach the shared memory space to process */
lumped = (float *) shmat(lumpid, lumpage, 0);
if (errno > 0)
        fprintf(stderr, " %s\n", sys_errlist[errno]);

for (i=0; i*nunits + noutputs; i++)
        lumped[i] = 0.0;

errno = 0;

/* attach arrays to shared memory addresses */
delta = &lumped[nunits];
/*delta = (float*) emalloc((unsigned)(sizeof(float)*nunits)); */
(void) install var("delta", Vfloat,(int *) delta, nunits, 0, SETVMENU);
error = &lumped[2*nunits];
/*error = (float*) emalloc((unsigned)(sizeof(float)*nunits)); */
(void) install var("error", Vfloat,(int *) error, nunits, 0, SETVMENU);
```

Figure 6-3
Shared Memory Allocation Code

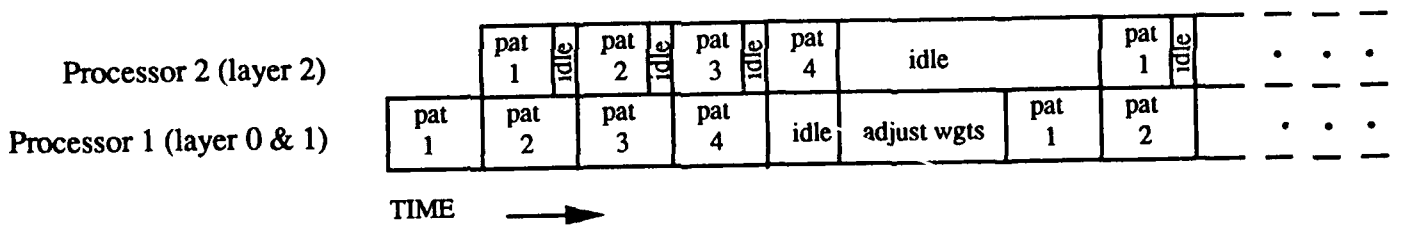


Figure 6-4

Two processor decomposition of a two-layer network with four patterns per epoch

Note idle times in processor 2. They are due to load imbalance between processor 1 and 2.

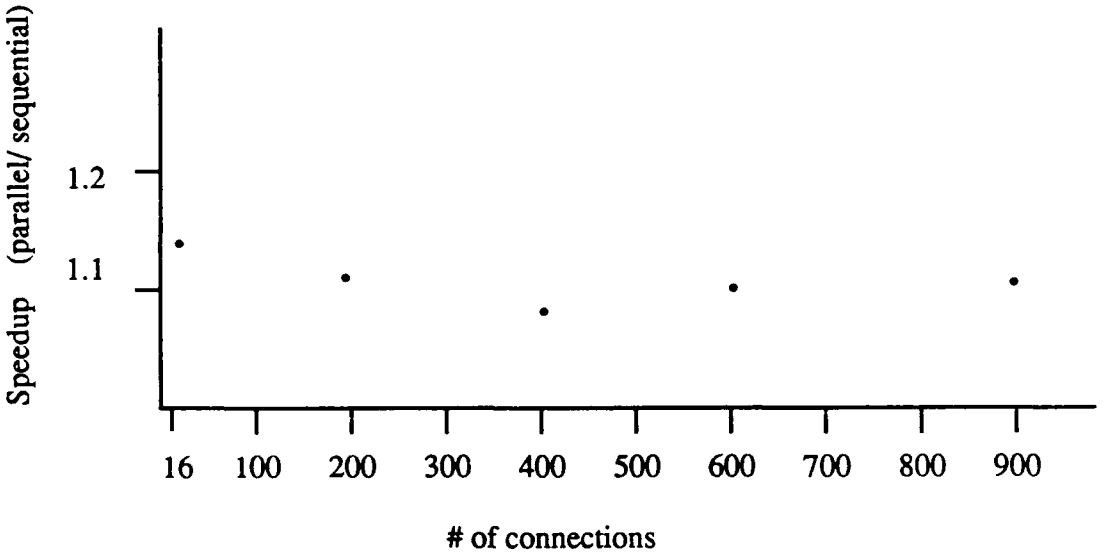


Figure 6-5
Speedup of PDP implementation of Layer Method for four patterns per epoch

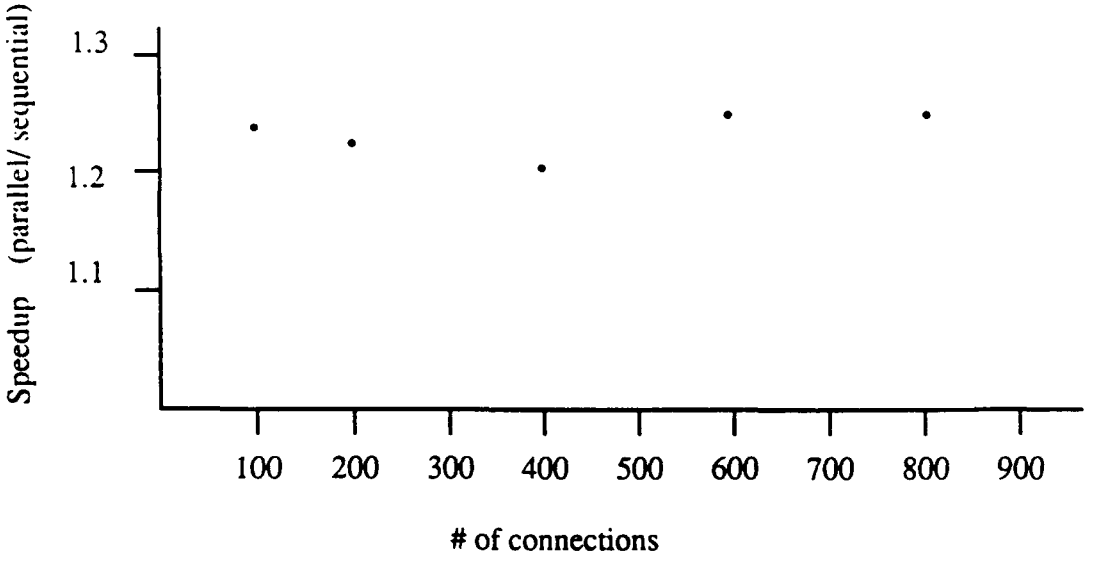


Figure 6-6
Speedup of PDP implementation of Layer Method for twenty patterns per epoch

Function	time to execute for entire network (msec)
read input/output vectors	0.1
Compute output	11.6
compute Logistic	0.2
compute error	12.4
Backprogagtion of errors	10.3
Adjust weights	22.1
Total	56.7

Table 6-2
Sequential verson Execution times for 103 network (single pattern)

Function	execution time (msec)
Processor 1 time for Table 6-1 functions	45.4
Processor 2 time for above functions minus weight adjusts	16.0
Total	61.4

Table 6-3
Layer Methold execution time for 103 network (single pattern)

cated why this was true. Execution times for the major functions that perform the training and learning algorithms for the sequential version of the PDP simulator are shown in table 6-2 for a network with 200 connections (103 network). These times are for a single pattern. Totalling up all the execution times except for the weight adjustments, results in 34.6 msecs.

Table 6-3 shows the execution times for a single pattern on the 103 network using the Layer method. Assuming the weight calculations for the sequential and layer methods are approximately the same, then processor one's execution time (excluding weight adjustments) is approximately 23.3 msecs. This, plus the time it takes processor two to do its calculations, results in a total execution time of 39.3 msec for the Layer method, which is approximately 5 msec slower than the sequential version.

If the Layer method was used with pattern training, longer execution times would result than if the network trained in a single processor. This is shown graphically in Figure 6-7. Parallel execution would take 61.4 msec per pattern whereas, sequential execution would only take 56.7 msec. Extrapolating this out for four patterns per epoch and twenty patterns per epoch would results in the numbers shown in figure 6-8. Thus, the speedup for a four pattern epoch on two processors would be:

$$\frac{160.5}{131.2} = 1.22$$

For a twenty pattern epoch, the speedup would be:

$$\frac{714.1}{504.1} = 1.41$$

This example clearly shows how the number of processors and the number of patterns per epoch affect the speedup for parallel implementations using the Layer method. Comparing the above example with the results obtained in

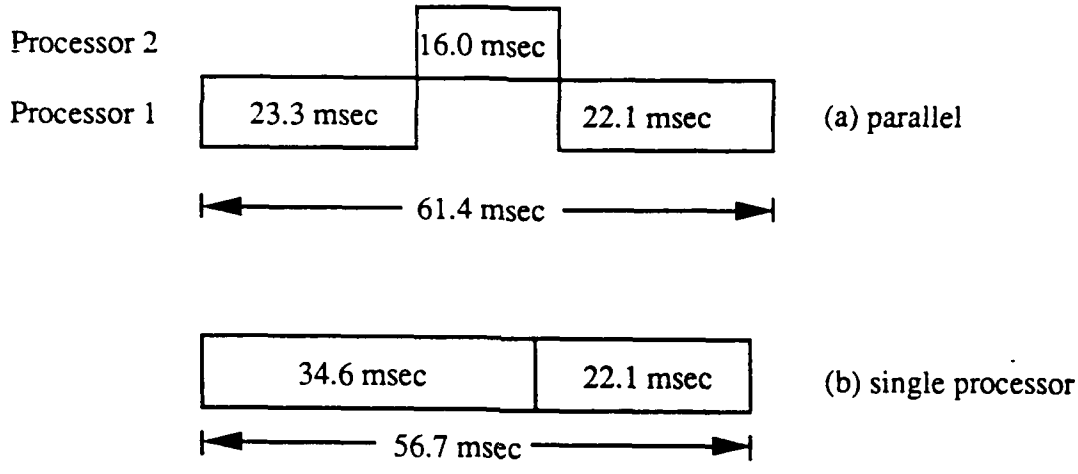


Figure 6-7
Processing times for one pattern on single and multiple processors

figures 6-5 and 6-6, and taking communication overhead into account, the speedups are in fair agreement with one another; 1.22 estimated, versus 1.12 actual for four patterns, and 1.41 estimated, versus 1.23 actual for twenty patterns.

From this analysis and the results shown, it is apparent that the Layer method is limited for several reasons. As pointed out above, one of its limitations is its dependance on the number of patterns per epoch. The only way to achieve a significant speedup is to have a large number of patterns to train on, and to train them on an epoch basis. Another limitation is that the amount

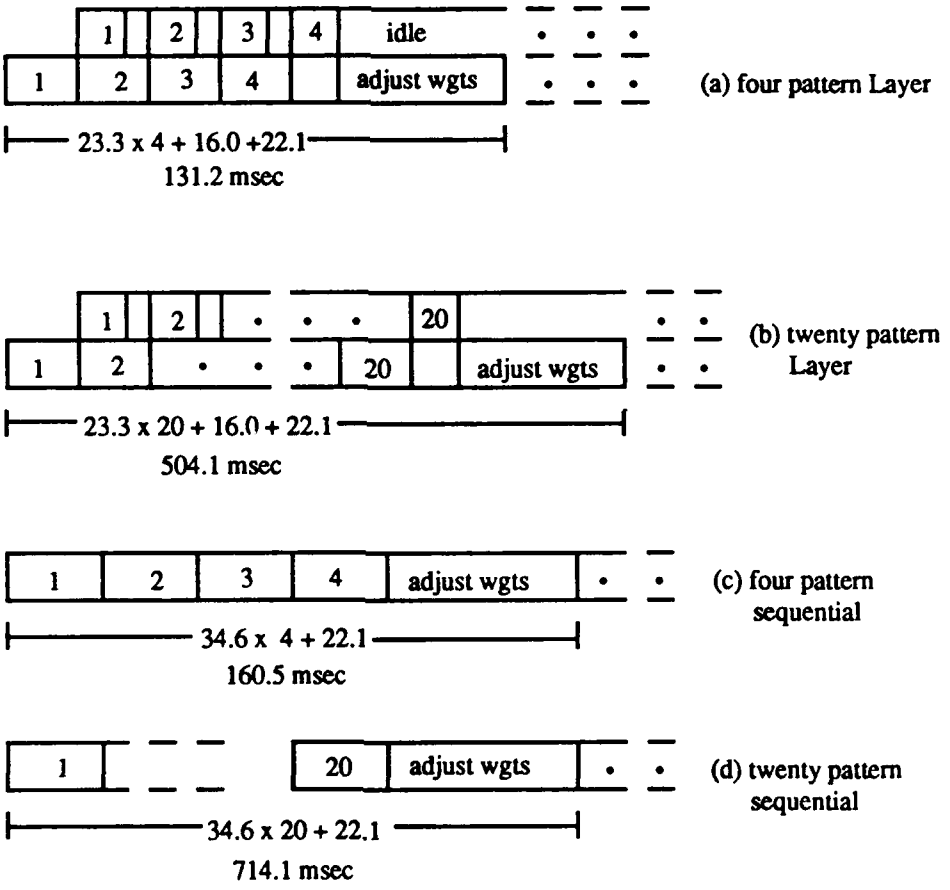


Figure 6-8
Processing times for four and twenty pattern epochs on single and multiple processors

of concurrency that can be realized with this method also depends on the number of processors available. Just as more patterns builds concurrency with this method, so do more processors. However, as indicated in figures 6-7 and 6-8, the amount of concurrency that can be realized with this method does not appear to be cost effective when one considers the extra coding involved and communication overhead. For these reasons, we felt that no further tests of this method were warranted on the PDP simulator, and that no tests need be conducted with the SDMO simulator.

Cross-layer Method Results and Analysis

The first test network I attempted using this method was the 424 network. This network has four input nodes, two hidden layer nodes, and four output nodes. The network was decomposed such that each processor was assigned exactly half of the nodes when a vertical slice was taken. In spite of the fact that substantial code changes were required to do this, I attempted to execute the 424 network with this methodology.

Unfortunately, this methodology was never successfully implemented due to problems with processor synchronization between layers. While it was possible to eventually synchronize the processors such that one processor would begin computing the inputs to a node after receiving all the data from the other processor, actual concurrent operation of the processors was minimal. Figure 6-9 better illustrates the problem.

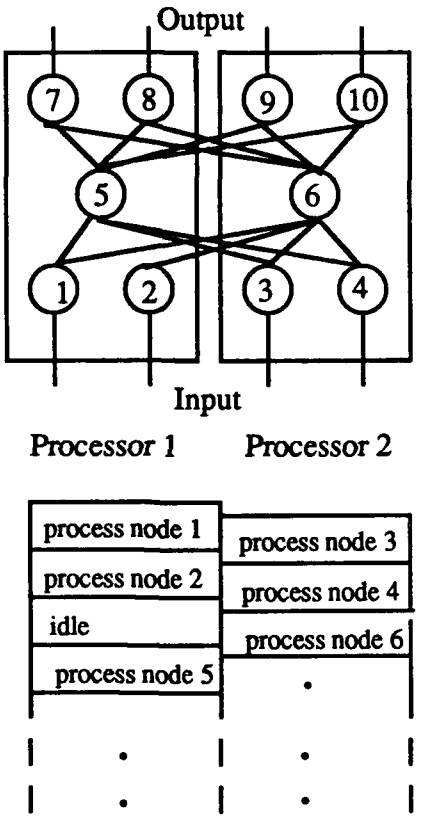


Figure 6-9
424 Cross Layer Decomposition

To avoid contention for the bus, processor one begins its execution slightly before processor two. This allows the results of nodes one and two to be presented to node six before nodes three and four do the same for node five. Unfortunately, before node five can begin computations, it needs the information from nodes three and four, which are purposely being delayed. Thus, node five's operation must be synched with the outputs of nodes three and four, to keep the results valid. It is this synchronization that causes most of the concurrency to be lost.

I attempted to let both processors start execution at the same time and let the operating system handle the contention for the bus. This was not successful either because the same overall effect occurred. Either node five was waiting for outputs from nodes three and four, or node six was waiting for outputs from nodes one and two.

Had the synchronization problem been solved, it was doubtful that any significant speedup would have been achieved. The main reason being, the overhead in terms of the coding required to make the vertical slices and to synchronize the processors was very high. Thus, it was doubtful that this overhead could have been offset by any gains in parallel execution.

Pipeline Method Results and Analysis

The decomposition and timing diagrams for the PDP and SDMO simulators are shown in figures 6-10 and 6-11. The diagrams illustrate how the

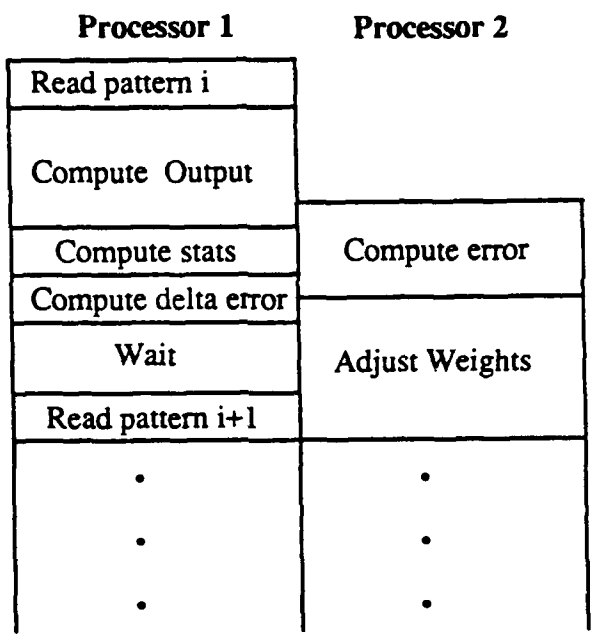


Figure 6-10
PDP Decomposition and timing diagram for Pipeline Method

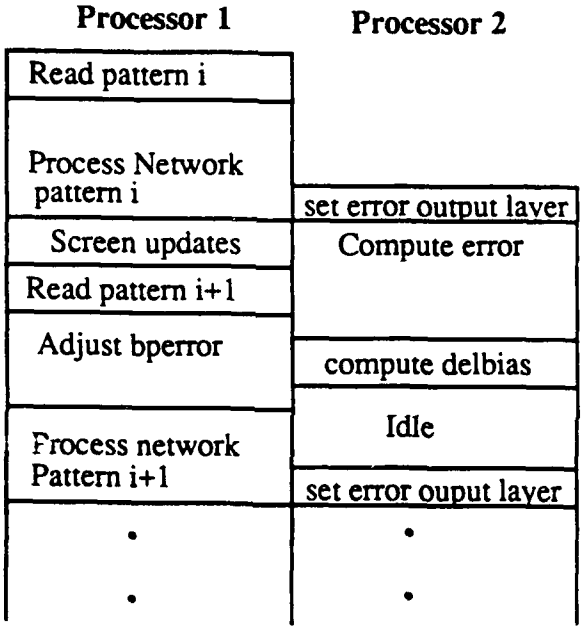


Figure 6-11
SMDO Decomposition and timing diagram for Pipeline Method

major functions of each simulator were decomposed, and the synchronization that was required. To better understand how this method was implemented, the SDMO simulator figure will be explained in detail.

There are seven functional blocks in the SDMO simulator that can be split up between the two available processors for parallel execution. Four of the blocks are assigned to processor one, while the other three are assigned to processor two. Processor one starts by reading in a pattern then immediately computing the resulting activation levels for each node.

As processor one begins these computations for the output layer, processor two, which has been idle up to this time, can begin setting the error array for the output nodes to zero (this must be done for every pattern). Immediately after this, processor two can begin to check the resulting error for the pattern and backpropagating the error to the other layers. While processor two is doing this, processor one begins updating information on the screen and reading in the next input pattern. By the time it finishes this, processor one can then begin adjusting the weights, starting at the output layer and working towards the input layer.

Even though processor two has not completed the error backpropagation, weight adjustment is possible, because the error propagation also starts at the output layer and works towards the input layer. Thus, by the time the weight adjustments start, the error for that layer has already been completed. While processor one is finishing the weight adjustments, processor two can then

make the necessary bias changes (delbias). Since the bias and weight adjustments are completely independent, no synchronization is required for these two functions.

The complete, modified source code for the PDP and SDM simulators for the Pipeline methodology are given in appendixes 3 and 4 respectively.

The results of the implementation of these simulators using the Pipeline method are given in figures 6-12 and 6-13. As shown in the figures the PDP implementation exhibited and average speedup of 1.67 and the SDM, an average speedup of 1.61. The average efficiencies of these implementations were:

$$\frac{\text{speedup}}{N} = \frac{1.67}{2} = 83\% \quad \text{PDP}$$

$$\frac{1.61}{2} = 80\% \quad \text{SDMO}$$

It should be pointed out that, due to the memory allocation scheme used in the SDMO simulator, and the operating system limitations placed on shared memory, only networks with up to 3200 connections could be simulated on the Masscomp with the SDMO simulator. While the operating system limits could have been increased to allow up to 10,000 connections for the SDMO simulator, a new system configuration file would have been required, and the operating system reconfigured. For these reasons, it was decided to limit the SDMO simulator to 3,200 connections.

It was interesting to note that the two simulators produced about the same average speedups. While doing the network decompositions, it was ap-

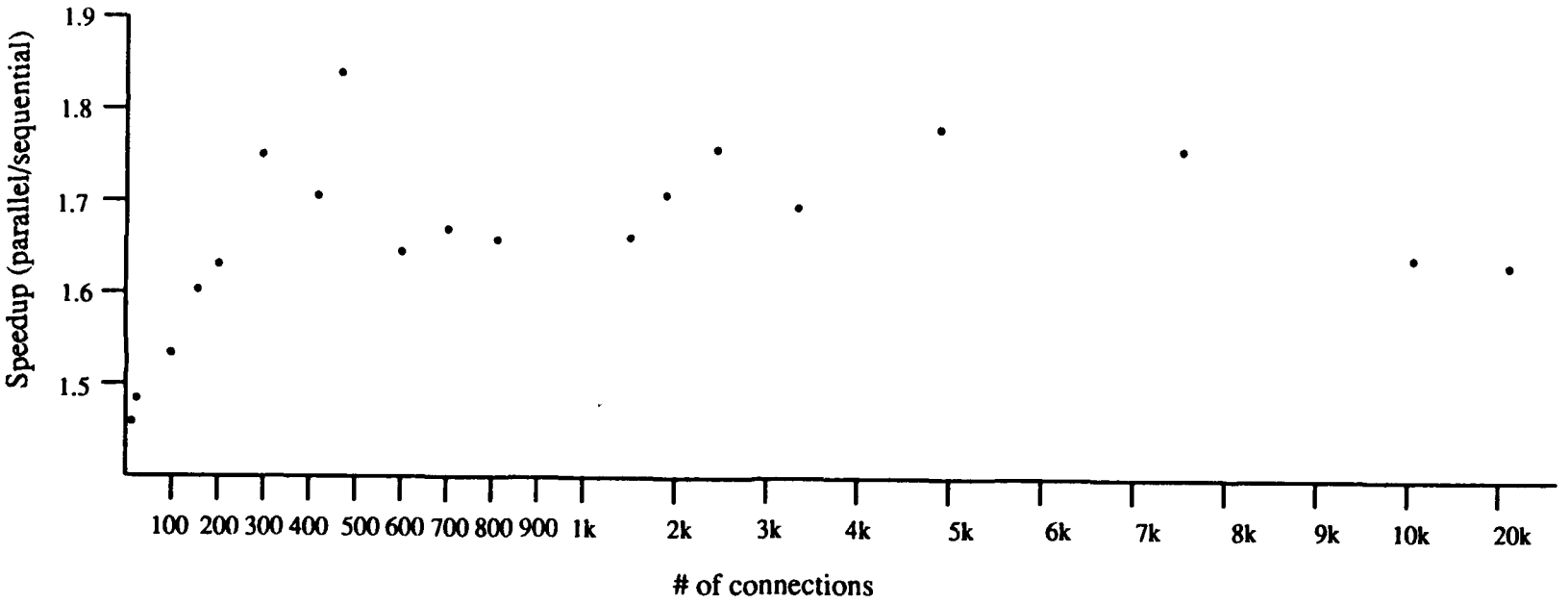


Figure 6-12

PDP Speedup with Pipeline Method

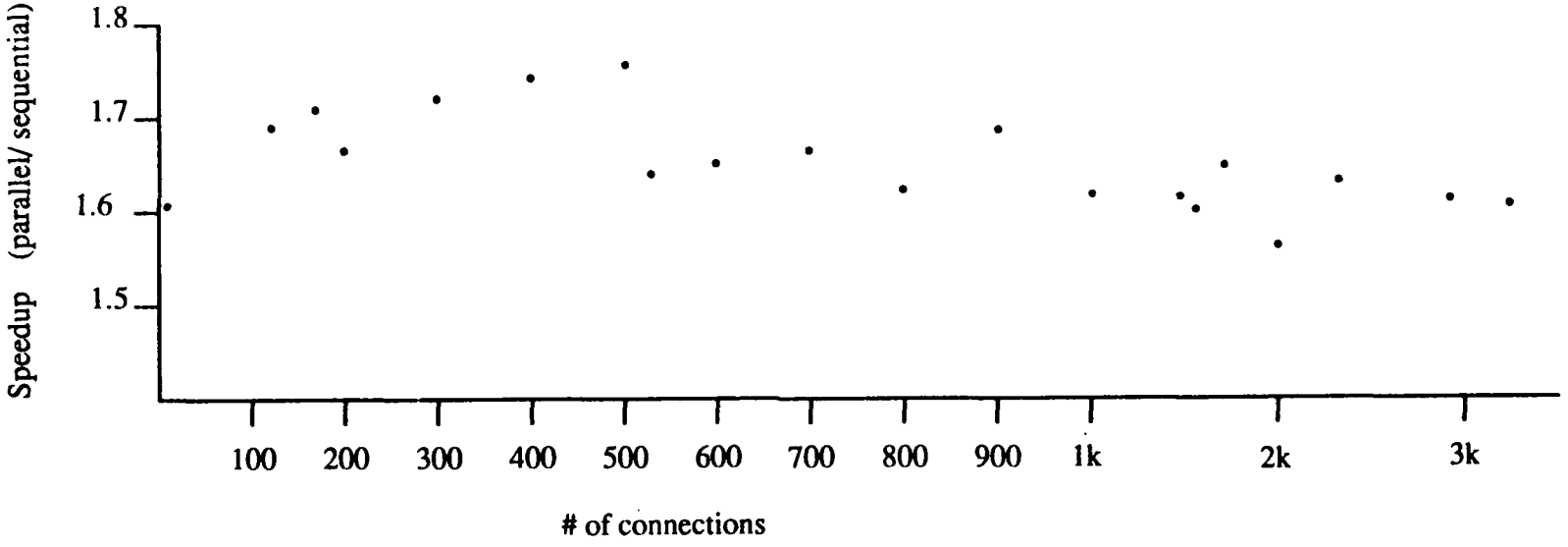


Figure 6-13

SDMO Speedup with Pipeline Method

parent that the PDP code was more effecient than the SDMO code. We believed that these differences would reflect in the speedup that each simulator could achieve.

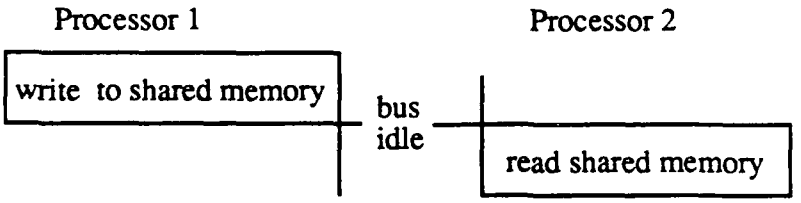
The PDP code was very succinct and well written. Nodes were numbered from 1 to N, where N was the total number of nodes in the network. This meant that only a single DO loop was required to perform a lot of the calculations. The SDMO simulator, on the other hand, was not as effecient. The nodes were numbered from 0 to l-1, where l is the number of nodes in a particular layer. Thus, in multilayer networks, one had to know the layer number as well as the node number to uniquely identify a particular node. This also meant, that instead of one DO loop for a calculation, several DO loops were required, depending on the number of layers in the network.

In spite of these differences, both simulators produced similar speedups. Based on these results, it appears as though speedup is more a function of how well an algorithm is decomposed into parallel blocks and not how well the algorithm is coded.

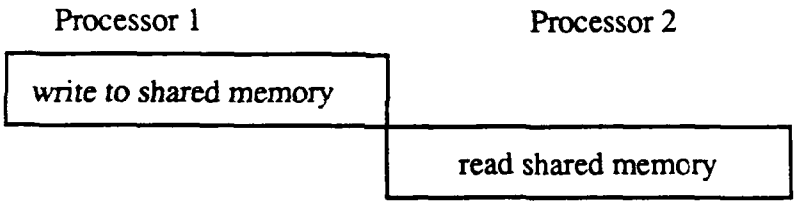
The "peaking" of the speedup shown in figures 6-12 and 6-13 was due to two factors. In networks with a small number of connections, speedup was inhibited because the staggering of the start times between processors was optimized for larger networks. For example, in the back propagation of errors and the adjustment of weight functions, the weight adjustments were not allowed to start until at least ten nodes had their errors calculated. In small net-

works, with say twenty nodes, this meant that weight adjustments could not start until half of all the error corrections were made. In the larger networks however, this provided enough "buffer" that the weight adjustments would not "overtake" the backpropagation of errors, causing erroneous weight adjustments.

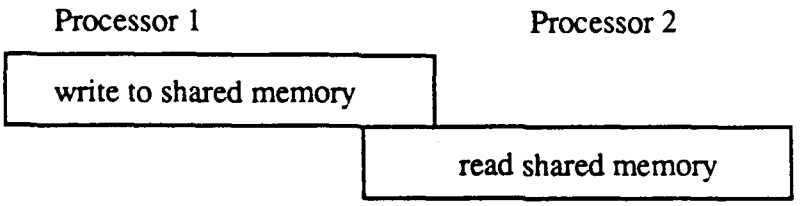
At the other end of the scale, where there were a large number of connections, the performance declined due to communication overhead. This was mainly due to the sheer volume of data that was being written into and read out of shared memory by each processor. Consider figure 6-14 as an example of what happens as the number of connections grows. In networks with a small number of connections, all the shared data can be read into and written out of shared memory in a few cycles. The time between when the data is written in and when it required is such that there is some idle time. This idle time is determined by the staggering of the functions on the different processors. As the number of connections grow, the idle time is decreased because it takes more time to accomplish the shared memory read and writes of all the required data. Eventually, as shown in figure 6-14b, the size of the network has grown to a point where there is no idle time left. Further increases in network size, then cause contention for the bus, thus performance begins to suffer (figure 6-14c).



(a) small network with memory read and write accomplished within processor stagger.



(b) Network size increased to the point where there is no idle bus time between processor 1's write and processor 2's reading of that data.



(c) Further increase in network size causes read from processor 2 to overlap with write from processor 1.

Figure 6-14
Network size's impact on centention of shared bus for memory reads and writes

Hybrid Epoch-Pattern Method Results and Analysis

The decomposition and timing diagrams for the PDP and SDMO simulators are shown in figures 6-15 and 6-16 respectively. These figures illustrate how pseudo epoch of two patterns, in a four pattern training set were implemented. Each processor does all the computations for its own patterns and runs asynchronously until weight adjustments are required. As shown, processor one performs the weight adjustments while processor two reads in the next two patterns, then waits for processor one to finish.

Notice in both figures, that the patterns are read by only one processor. This was not the original intent for this methodology; it was forced on by the way the C programming language handles formatted files. In C, when scanning formatted files, an internal pointer is used to index to the next value to be written into, or read out of the file. Even if the formatted file is in shared memory, each process with access to the file has its own copy of the internal pointer, that is, the pointer cannot be shared. The reason for this is that the pointer is associated with a process id and not the file itself. This way multiple users can access the same file, and do not have to worry about someone else moving their internal index.

An alternative solution to this pattern reading problem could have been to have each processor skip over patterns it does not need to process. This alternative was rejected however, because the cyclic nature of the pattern group-

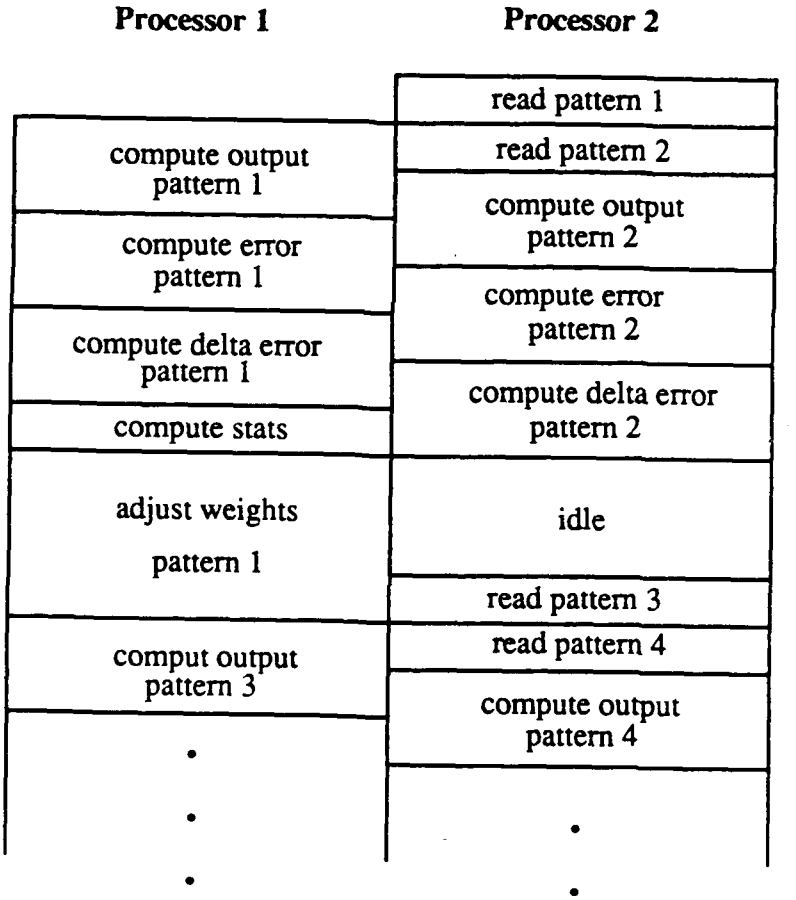


Figure 6-15
PDP decomposition and timing diagram for Hybrid Epoch-pattern approach

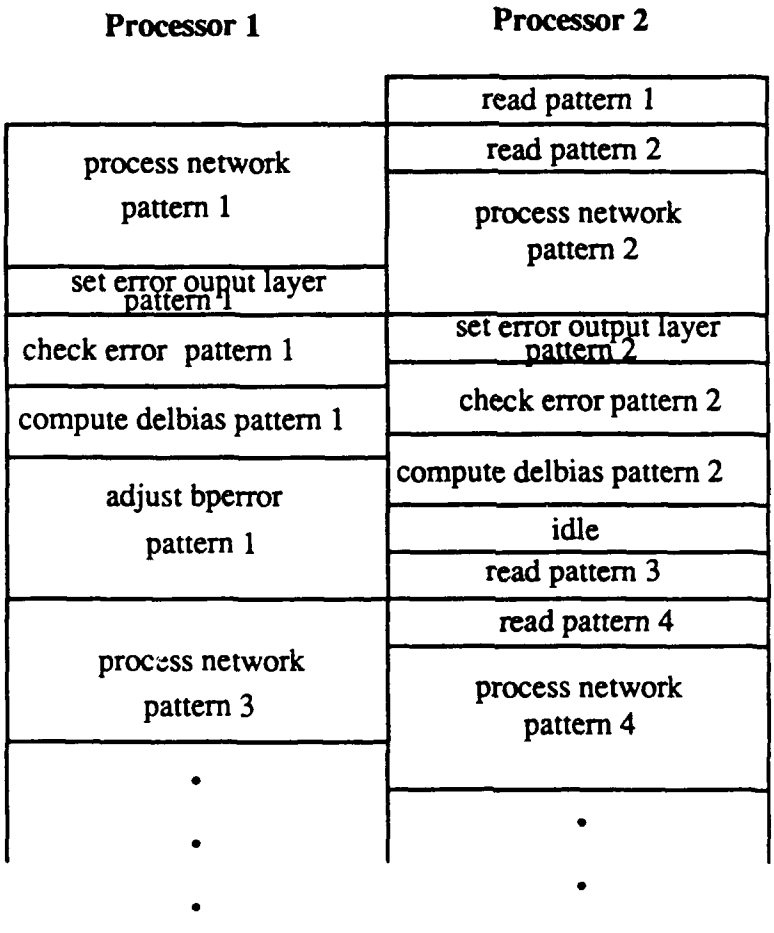


Figure 6-16
SDMO decomposition and timing diagram for Hybrid Epoch-pattern approach

ings (explained in chapter 5) would have required an excessive amount of code to keep the processors reading the proper set of patterns.

The complete, modified source code for the PDP and SDMO simulator implementations using the Hybrid method are given in appendixes 5 and 6 respectively.

The results of these implementations are shown in figures 6-17 through 6-24. The figures show the speedup for a pseudo epoch of two, for a four pattern training set, and a pseudo epoch of five, for a ten pattern training set. These pseudo epochs were compared against the implementation of the test networks on a single processor using both the pattern training method and the epoch training method.

As shown in the figures, an overall average speedup between the hybrid method and the two classical training methods was 2.0. Considering communication overhead and the execution times associated with the extra "parallelizing" code, a speedup of 2 or better would seem impossible. However, three conditions existed that made this possible.

Firstly, the weight adjustments and the reading of the patterns were performed concurrently, which would allow for a small decrease in execution time. Secondly, the weight adjustments were only performed every two patterns in the four pattern training set, and every five patterns in the ten pattern training set (as opposed to pattern training's four and ten weight adjustments respectively). Thirdly, the hybrid method tended to converge to a solution in a slightly

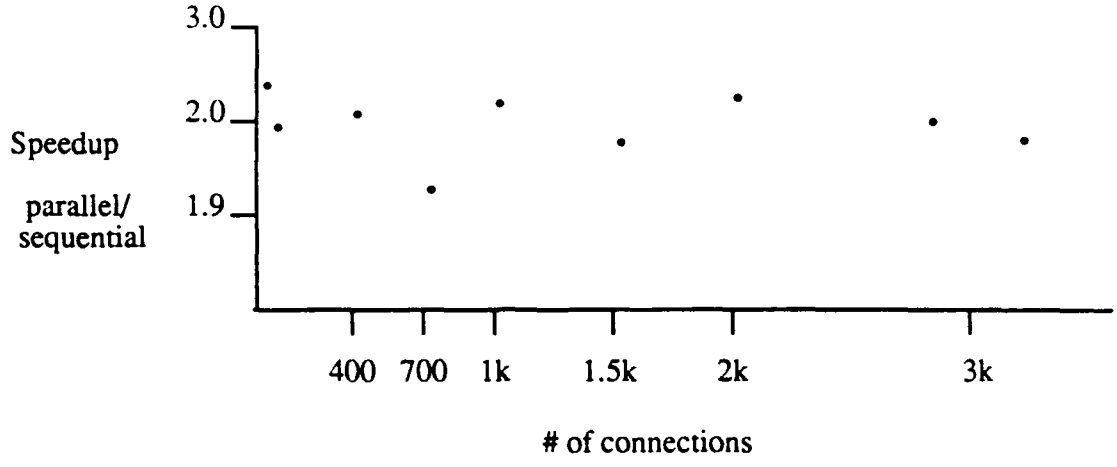


Figure 6-17
PDP Speedup with Hybrid method for four patterns vs. Sequential Pattern training

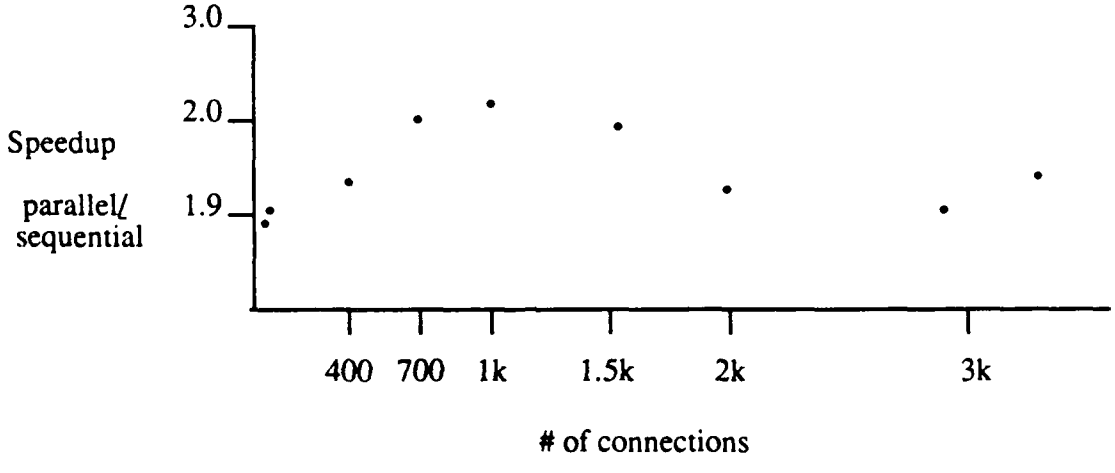


Figure 6-18
SDMO Speedup with Hybrid method for four patterns vs. Sequential Pattern training

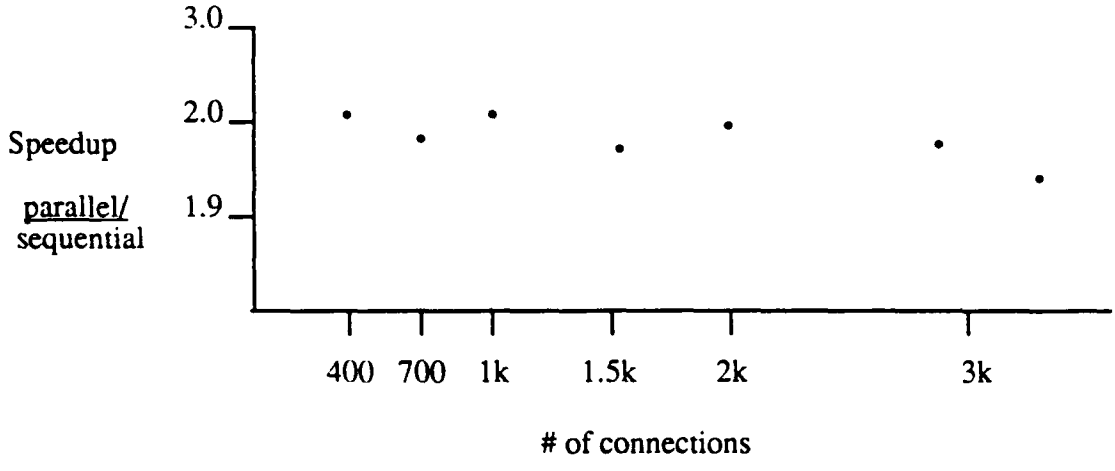


Figure 6-19
PDP Speedup with Hybrid method for ten patterns vs. Sequential Pattern training

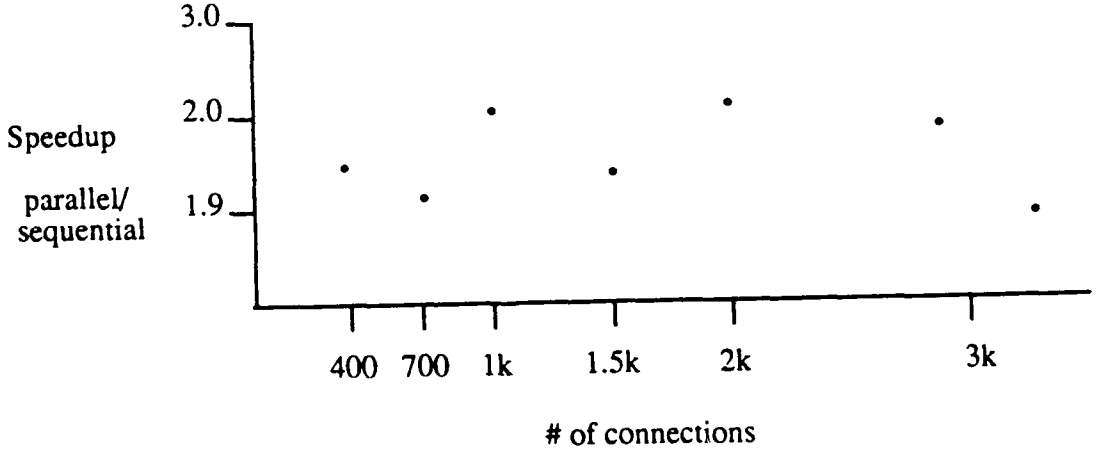


Figure 6-20
SDMO Speedup with Hybrid method for ten patterns vs. Sequential Pattern training

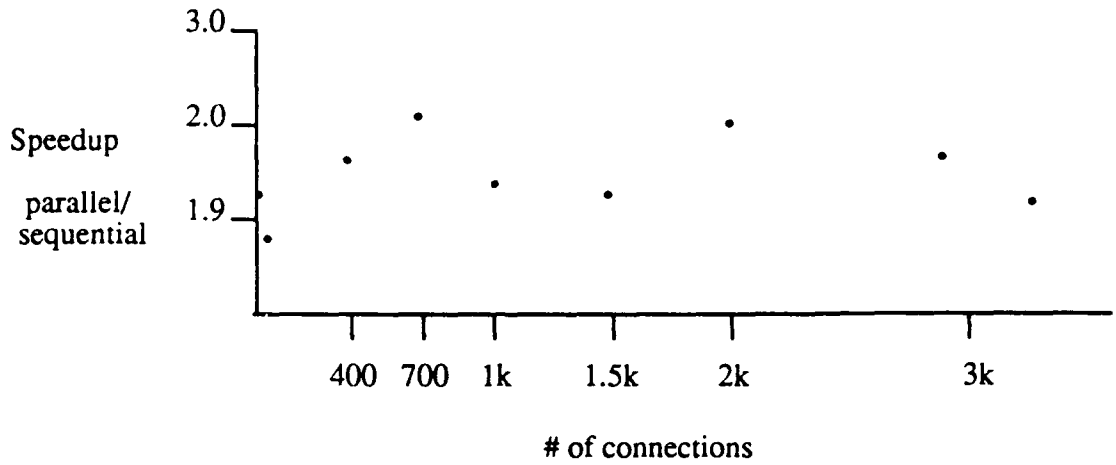


Figure 6-21
PDP Speedup with Hybrid method for four patterns vs. Sequential Epoch training

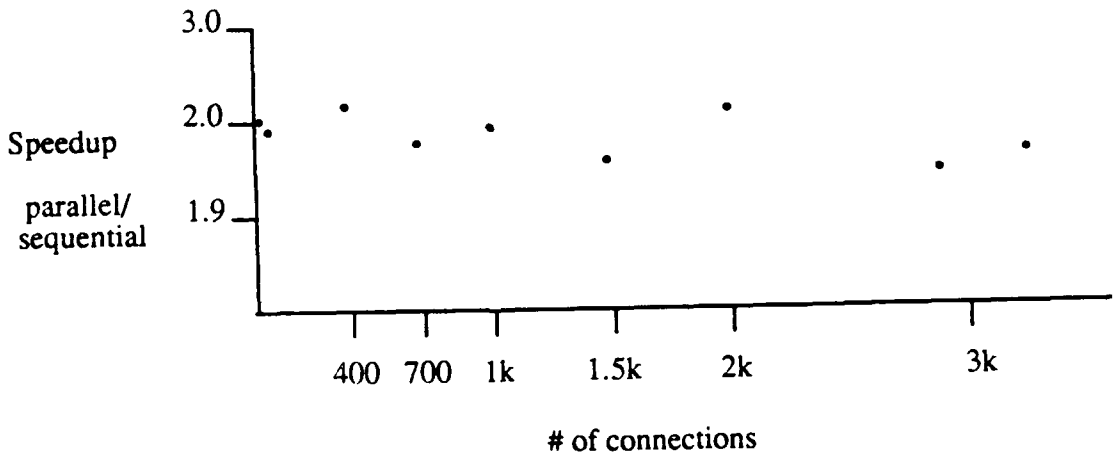


Figure 6-22
SDMO Speedup with Hybrid method for four patterns vs. Sequential Epoch training

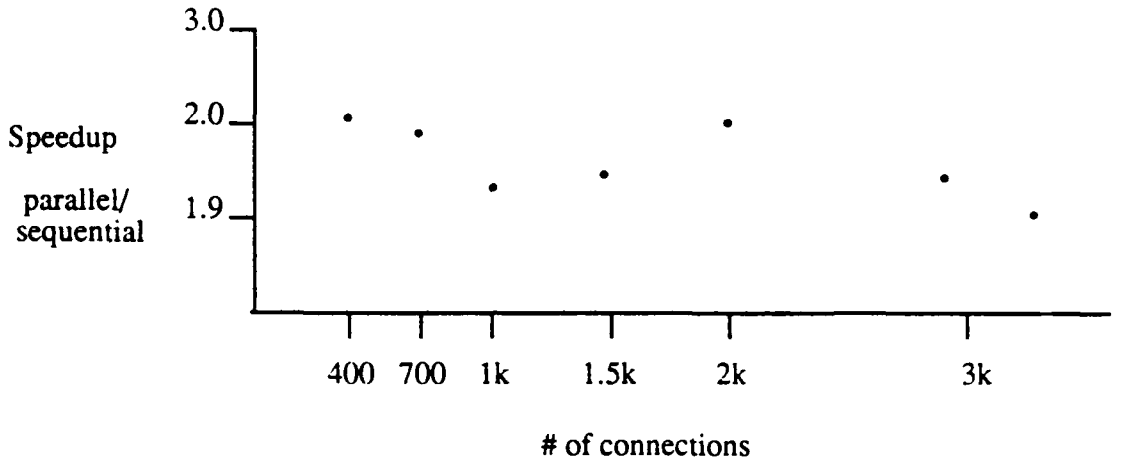


Figure 6-23
PDP Speedup with Hybrid method for ten patterns vs. Sequential Epoch training

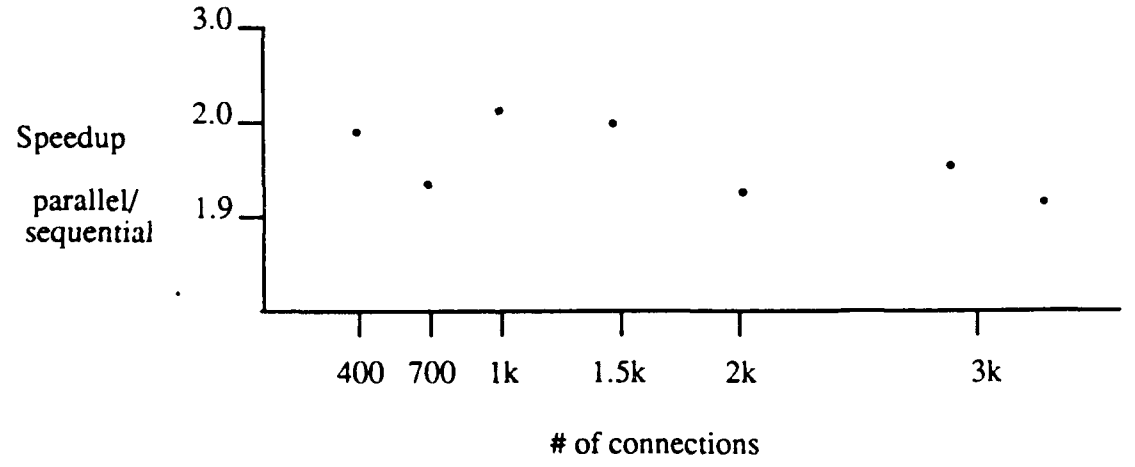


Figure 6-24
SDMO Speedup with Hybrid method for ten patterns vs. Sequential Epoch training

smaller number of cycles than the epoch method (differences were typically ten or less).

Thus, when compared to pattern training, the time gained by performing less weight adjustments and the parallel operations, offset the overhead incurred by executing the network on multiple processors. When compared to epoch training, near perfect speedups were possible because of the parallel operations and the slightly smaller number of cycles to convergence.

The number of data points collected for this methodology is somewhat smaller than those collected in the Pipeline method. The reason for this was because of the difficultly in getting all three training methods (i.e. single processor pattern and epoch methods and multiprocessor hybrid method) to converge *to the exact same solution.* In all cases, the resulting solutions were correct but they were not always the same solutions for all three methods.

The reason for the different solutions was two fold. First, there are multiple sets of correct weights that will process all the patterns properly. Slight variances in where the network falls on the error surface could therefore cause different solutions to be sought out. Secondly, each training method adjusts the weights after a different number of patterns. Thus, they might be in different places on the error surface after going through the same number of patterns, and could therefore head towards different minima solutions.

To help "coax" the training methods towards the same solution, I "biased" the weights. Normally, the weights are initially randomized between -1

and 1, with each training method getting the exact same randomized weights. To bias the weights, I ran one training method (usually the pattern method) for fifty to one hundred epochs, then used the resulting weights as our initial set of weights for all three methods. In doing this, the network was hopefully far enough along the error surface that only a single solution would be sought, regardless of the training method. While this procedure was fairly successful, it significantly increased the amount of time it took to collect the data.

Summary

The four methodologies developed in chapter V were applied to two neural network simulators and executed on a coarse grain, bus oriented, multiprocessor computer system.

The Layer method showed a very small amount of speedup. This was primarily attributed to the concurrency limitations imposed by the number of patterns in the networks' training sets and the number of processors utilized. The Cross-layer method was never successfully implemented. While the fundamental concepts behind the methodology were sound, the synchronization between the processors could not be satisfactorily solved.

The Pipeline method resulted in an average speedup of 1.67, which corresponds to a 40% reduction in execution time over a single processor system. The simulators were successfully decomposed into functional blocks that could

be executed concurrently. Through the use of this technique and the staggering of dependant blocks, the resulting speedups were achievable.

The Hybrid Epoch-Pattern method was also successful with an average speed up of 2, which corresponds to a doubling of the processing speed. This method also showed that training on pseudo epochs is a viable alternative to the classic pattern and epoch training methods.

Chapter VII

Conclusions and Recommendations

General Conclusions

Neural network research and development has seen a resurgence over the past five years. Most of this research is being aided by the use of software simulations of neural networks being run on single processor computer systems. Unfortunately, these simulations are often extremely slow because of the sheer size of the networks and the large number of computations. As such, researchers are often limited in the size of the network that can be practically simulated.

While the eventual implementation of neural networks in hardware will help alleviate some of these problems, such implementations are neither affordable nor practical at this time. Besides, hardware implementations require their own extensive network simulations and validation of design before committing the network to a chip.

With this in mind, the primary goal of this dissertation was to lay the foundation for the implementation of neural network technology on multiprocessor systems to increase network simulation speeds and size capabilities.

To attain this goal, I started by discussing the interrelationship of neural network topology, with network learning and training in Chapter II. I then detailed various multiprocessor architectures and their characteristic features in Chapter III.

In Chapter IV, a conceptual framework was presented based on the concepts of program decomposition, load balancing, communication overhead, and process synchronization. A set of metrics was also introduced to enable us to measure performance enhancements of multiprocessor implementations, and to analyze the effects of load balancing, communication overhead and synchronization.

In Chapter V, I developed four methodologies for the implementation of neural network simulators on multiprocessor systems: the Layer, Cross-layer, Pipeline, and Hybrid Epoch-Pattern methods. The first two methodologies were based on the topological aspects of neural networks, whereas the second two were based on their functional aspects.

The practical application of these methods were demonstrated in Chapter VI, where two neural network simulators were implemented on a multiprocessor system using these four methods. I demonstrated the implementation of networks ranging from six to twenty thousand connections that resulted

in speedups averaging 1.66 and 2.0 for the Pipeline and Hybrid method respectively. These results proved that these techniques and their generalities enable the efficient multiprocessor implementation of highly complex massive neural networks with acceptable execution times.

Recommendations

With the primary objectives of this dissertation fulfilled, we can now examine some areas of future research this work has generated.

One of the natural follow-ons to this research is to build a C compiler to automate the decomposition process. I envision this compiler being used as a pre-processor, where several test patterns are presented to enable the compiler to identify all parallel blocks of code and shared memory variables. Once this is accomplished, the compiler would recompile the program, making the necessary processor and shared memory assignments. Work in this area has already been discussed with Wright State University for several Masters thesis projects.

A second, similar area of research involves using compilers to seek out any and all concurrency, whether it be at the block level or DO loop level. I believe this would allow the maximum flexibility in the assignment of tasks to the various processors and help optimize load balancing.

An area of research the Wright Research and Development Center (WRDC) Advanced Information Processing and Technology Branch at Wright

Patterson AFB Ohio would like to begin, is the implementation of the Pipeline method on several Adaptive Resonance Theory network simulators. They wish to use a SUN Workstation as a host processor to thirty transputers, to implement these networks. This would further expand the knowledge base on multiprocessor implementation techniques.

From the standpoint of direct applications, McDonnel Douglas is very interested in the Layer, Pipeline, and Hybrid methodologies. They are currently working on developing several neural network simulators using the backpropagation algorithm. Their ultimate goal is to design a hardware realizable neural network for real time multispectral image fusion. They wish to use the techniques and methods we have developed to aid in the research and development phases of their backpropagation neural network simulations.

Appendix 1

Test Neural Networks

Name	# of inputs	# of outputs	Total	Layer Arrangement	# of connections
XOR	2	1	6	2/2/1	5
424	4	4	10	4/2/4	16
102	10	10	20	10/10	100
151	10	10	25	10/5/10	100
103	10	10	30	10/10/10	200
1881	10	10	36	10/8/8/10	224
104	10	10	40	10/../10	300
1151	10	10	35	10/15/10	300
2020	12	10	44	12/10/12/10	360
105	10	10	50	10/../10	400
121	10	10	40	10/20/10	400
106	10	10	60	10/../10	500
1551	10	10	50	10/15/15/10	525
131	10	10	50	10/30/10	600
2125	20	5	60	20/15/20/5	700
2211	20	10	60	20/20/10/10	700
1221	10	10	60	10/20/20/10	800
231	20	10	60	20/30/10	900
2212	20	20	75	20/20/15/20	1,000
2252	20	20	65	20/25/20	1,000
1331	10	10	80	10/30/30/10	1,500
242	20	20	80	20/40/20	1,600
13231	10	10	120	10/30/20/30/10	1,800
252	20	20	90	20/50/20	2,000
2422	20	20	100	20/40/20/20	2,000
1441	10	10	100	10/40/40/10	2,400
2441	20	10	110	20/40/40/10	2,800
2442	20	10	120	20/40/40/20	3,200
14441	10	10	140	10/40/../10	4,000
24442	20	20	160	20/40/../20	4,800
5555	50	50	200	50/50/50/50	7,500
5105	50	50	200	50/100/50	10,000
111	100	100	300	100/100/100	20,000

Appendix 2

C Source Code for Stats Program

```
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#define RUSAGE_SELF     0
#define RUSAGE_CHILDREN -1
struct rusage  rbuff;

rstats (fp)
FILE *fp;
{  /* begin rstats */
 printf("---------beginning of stats---------\n \n");
getrusage (RUSAGE_SELF, &rbuff);
fprintf (fp,"user time used\t\t= %ld sec  %ld usec\n",
rbuff.ru_utime.tv_sec, rbuff.ru_utime.tv_usec);
fprintf (fp,"system time used\t= %ld sec  %ld usec\n",
rbuff.ru_stime.tv_sec, rbuff.ru_stime.tv_usec);
fprintf (fp,"max set size in Kbytes \t= %d\n",
rbuff.ru_maxrss);
fprintf (fp,"shared memory size\t= %d\n", rbuff.ru_ixrss);
fprintf (fp,"unshared data size\t= %d\n", rbuff.ru_idrss);
fprintf (fp,"shared stack size\t= %d\n", rbuff.ru_isrss);
fprintf (fp,"page reclaims\t\t= %d\n", rbuff.ru_minflt);
fprintf (fp,"page faults\t\t= %d\n", rbuff.ru_majflt);
fprintf (fp,"swaps\t\t\t= %d\n", rbuff.ru_nswap);
fprintf (fp,"block inputs\t\t= %d\n", rbuff.ru_inblock);
fprintf (fp,"block outputs\t\t= %d\n", rbuff.ru_oublock);
fprintf (fp,"voluntary switches\t= %d\n", rbuff.ru_nvcsw);
fprintf (fp,"involuntary switches\t= %d\n",
rbuff.ru_nivcsw);

printf("----------end of stats----------\n \n");

} /* end of rstats */

rtime (fp)
FILE *fp;
{  /*begin rtime*/
getrusage (RUSAGE_SELF, &rbuff);
fprintf (fp,"user time used\t\t= %ld sec  %ld usec\n",
rbuff.ru_utime.tv_sec, rbuff.ru_utime.tv_usec);

fprintf (fp,"system time used\t= %ld sec  %ld usec\n",
rbuff.ru_stime.tv_sec, rbuff.ru_stime.tv_usec);

} /* end of rtime */
```

```
rtd(utime,stime)
double      *utime,*stime;

{ /* begin rtd */
getrusage (RUSAGE_SELF, &rbuff);
*utime = (rbuff.ru_utime.tv_sec) +
(rbuff.ru_utime.tv_usec)/((double)1.0e6);
*stime = (rbuff.ru_stime.tv_sec) +
(rbuff.ru_stime.tv_usec)/((double)1.0e6);

} /* end of rtd */

double rtd_t()
{ /* begin double */
getrusage (RUSAGE_SELF, &rbuff);
return((rbuff.ru_utime.tv_sec) +
(rbuff.ru_utime.tv_usec)/1.0e6
+ (rbuff.ru_stime.tv_sec) +
(rbuff.ru_st      _usec)/1.0e6);

} /* end of double */

rtl(utime_s,utime_u,stime_s,stime_u)
long*utime_s,*utime_u,*stime_s,*stime_u;

{ /* begin rtl */
getrusage (RUSAGE_SELF, &rbuff);
*utime_s = rbuff.ru_utime.tv_sec;
*utime_u = rbuff.ru_utime.tv_usec;
*stime_s = rbuff.ru_stime.tv_sec;
*stime_u = rbuff.ru_stime.tv_usec;

} /* end of rtl */
```

Appendix 3

Modified PDP Source Code for Pipeline Method

GENERAL.C

```
/* This file is part of the PDP software package.  Copyright 1987 by
James L. McClelland and David E.  Rumelhart. Please refer to licensing
information in the  file license.txt, which is in the same directory
with this source file and is included here by reference.*/

/* general.c
     Some general functions for PPD-pc package.
     First version implemented by Elliot Jaffe.
     Date of last revision:  8-12-87/JLM.
     Masscomp revisions:  30 Jan 89/ bob Bennington */

#include "general.h"
#include "command.h"
#include "variable.h"
#include <signal.h>

#ifdef MSDOS
#include <memory.h>   /* for memcpy() in erealloc in general.c */
#include <process.h>  /* for system() in do_exec in command.c */
#endif

FILE * in_stream = stdin;
int     Interrupt_flag = 0;
int  single_flag = 0;
int  step_size;
int  random_seed;
char step_string[STRINGLENGTH];
struct Command_table   *Command;

extern int  dump_template ();
extern int  clear_display ();
extern int  update_display ();
extern int  redisplay ();
extern int  dc_io ();
extern int  do_network ();
extern int  do_system ();
extern int  do_command ();
extern int  do_comfile ();
extern int  do_exec ();
extern int  set_log ();
extern int  run_fork();
extern float *wds;
extern int wdsid;
```

```
extern int lumpid;
extern float *lumped;
extern int *misc;
extern int miscid;

extern float *dwts;
extern int dwtsid;
extern float *wtss;
extern int wtssid;
extern int fnwtid;
extern int *fnwt;
extern float *wds;
extern float **wtwed;
extern int wtwedid;

int_handler() {
    int     int_handler ();

    (void) sig[illegible]INT, int_handler);
    Interrupt_flag = 1;
}

#ifdef MSDOS
char *index(somestring,somechar)
char *somestring;
char somechar;
{
    return strchr(somestring,somechar);
}
#endif

char   *emalloc (n)          /* check return from malloc */
unsigned    n;
{
    char   *p,
           *malloc ();

    p = malloc(n);
    if (p == 0)
     put_error("out of memory");
    return p;
}

char   *erealloc (ptr,oldsize,newsize)
/* check return from realloc*/
char *ptr;
unsigned    oldsize;
unsigned    newsize;
{
#ifndef MSDOS
    char   *realloc ();
    char   *p;
```

```
    p = realloc(ptr,newsize);
    if (p == 0)
     put_error("out of memory");
    return p;

#else (if MSDOS)

    char *malloc();
    char *p;

    p = malloc(newsize);
    if (p == 0)
        put_error("out of memory");
    if (ptr && p) {
     memcpy(p, ptr, oldsize);
     free(ptr);
    }
    return p;

#endif MSDOS
}

calc_page(size)     /* added 7 apr 89 RWB */
int size   /* to calc amount of space needed for */
{       /* shared memory in increments of 4K pages */
int inc, pg, endof, xtra, nuend;
inc = 1;
pg = 4096;
endof = sbrk(0);
while (pg<= endof){
     pg = 4096 * inc;
     ++inc;
}
xtra = (pg - endof) + size;
nuend = sbrk(xtra);
return(pg);
}

startsame(s1, s2)      /* does s1 start the same as s2? */
char   *s1,
       *s2; {
    while (*s1 && *s2) {
     if (*s1++ != *s2++)
         return(0);
    }
    if(*s1 && !*s2) /* if s1 is longer than s2 it should fail */
       return(0);
    return(1);
}
```

```
char   *strsave (s)
char   *s;
{
    char   *p,
           *emalloc ();

    if ((p = emalloc((unsigned)(strlen(s) + 1))) != NULL)
     (void) strcpy(p, s);
    return(p);
}


randint(low, high)
 int  low,high; {
    int     answer;
    float   randf;
    int     range;

    randf = rnd();
    range = high - low + 1;
    answer = randf * range + low;
    return(answer);
}

quit() {
   int r;
   char * str;
   str = get_co:  .("Quit program? -- type y to confirm:  ");

   if (str && str[0] == 'y') {
         end_display();
           /* added 7 apr 89 detaches shared */
             /* memory  and deallocates its storage- RWB */
         shmdt(misc);
            shmdt(lumped);
         shmdt(fnwt);
            shmdt(wtss);
         shmdt(dwts);
         shmdt(wds);
         shmdt(wtwed);
            shmctl(fnwtid, IPC_RMID,0);
            shmctl(miscid, IPC_RMID,0);
            shmctl(lumpid, IPC_RMID,0);
            shmctl(wtwedid, IPC_RMID,0);
            shmctl(dwtsid, IPC_RMID,0);
            shmctl(wtssid, IPC_RMID,0);
            shmctl(wdsid, IPC_RMID,0);

     exit(0);
   }
   else
      return(CONTINUE);
```

```
}
stats() {                       /* this function clears the screen then
prints out */
clear_display();  /*the stats package. It also returns the curser */
io_move(5,0);  /* back where it belongs so the command line will be */
io_refresh(); /* in the proper position  --31Jan 89 bob bennington */
rstats(stdout);
io_move (0,0);
io_refresh();
return(CONTINUE);

}


set_step() {
    char old_step_string[STRINGLENGTH];
    struct Variable *vp, *lookup_var();

    strcpy(old_step_string,step_string);

    vp = lookup_var("stepsize");
    change_variable("stepsize",vp);

    if (startsame(step_string,"nepochs"))
strcpy(step_string,"nepochs");
    else if (startsame(step_string,"epoch"))
strcpy(step_string,"epoch");
    else if (startsame(step_string,"pattern"))
strcpy(step_string,"pattern");
    else if (startsame(step_string,"ncycles"))
strcpy(step_string,"ncycles");
    else if (startsame(step_string,"cycle"))
strcpy(step_string,"cycle");
    else if (startsame(step_string,"update"))
strcpy(step_string,"update");
    else if (startsame(step_string,"default"))
                        strcpy(step_string,Default_step_string);
    else {
     strcpy(step_string,old_step_string);
     return(put_error("urecognized stepsize -- size not changed."));
    }
    set_stepsize();
    return(CONTINUE);
}

set_stepsize() {
    if (strcmp(step_string,"update") == 0) step_size = UPDATE;
    else if (strcmp(step_string,"cycle") == 0) step_size = CYCLE;
    else if (strcmp(step_string,"ncycles") == 0) step_size = NCYCLES;
    else if (strcmp(step_string,"pattern") == 0) step_size = PATTERN;
    else if (strcmp(step_string,"epoch") == 0) step_size = EPOCH;
    else if (strcmp(step_string,"nepochs") == 0) step_size = NEPOCHS;
```

```
}

init_general() {
    extern int     int_handler ();

    Interrupt_flag = 0;
    strcpy(step_string,Default_step_string);
    set_stepsize();
    init_commands();
    (void) signal(SIGINT, int_handler);
    (void) install_command("?", do_help, 0, 0);
    (void) install_command("disp/", do_command, BASEMENU, (int *)
DISPLAYMENU);
    (void) install_command("opt/",  do_command, DISPLAYMENU, (int *)
DISPLAYOPTIONS);

    (void) install_command("exam/", do_command, BASEMENU, (int *)
SETMENU);
    (void) install_command("get/",  do_command, BASEMENU, (int *)
GETMENU);
    (void) install_command("save/", do_command, BASEMENU, (int *)
SAVEMENU);
    (void) install_command("set/",  do_command, BASEMENU, (int *)
SETMENU);
    (void) install_command("config/", do_command, SETMENU, (int *)
SETCONFMENU);
    (void) install_command("env/",  do_command, SETMENU, (int *)
SETENVMENU);
    (void) install_command("mode/", do_command, SETMENU, (int *)
SETMODEMENU);
    (void) install_command("param/",do_command, SETMENU, (int *)
SETPARAMMENU);
    (void) install_command("state/", do_command, SETMENU, (int *)
SETSVMENU);
    (void) install_command("clear", clear_display, BASEMENU, 0);
    (void) install_command("do",  do_comfile, BASEMENU, 0);
    (void) install_command("log",   set_log, BASEMENU, 0);
    (void) install_command("quit",  quit, BASEMENU, 0);
    (void) install_command("run", do_exec, BASEMENU, 0);
    (void) install_command("stats", stats, BASEMENU, 0); /* added 30
Jan89 by bob bennington */
/*  (void) install_command("srand", random_seed, BASEMENU, 0); */
    (void) install_command("state", redisplay, DISPLAYMENU, 0);
    (void) install_var("seed", Int, (int *) & random_seed, 0,
0,SETPCMENU);
    (void) install_var("single", Int, (int *) & single_flag, 0,
0,SETPCMENU);
    (void) install_var("stepsize", String, (int *) step_string,0,
0,NOMENU);
    (void)   install_command("stepsize",set_step,SETPCMENU,(int   *)
NULL);
}
```

```
#ifdef MSDOS
sleep(n_sec)
int n_sec;
{
    int i,j;
    for (i = 0; i < (n_sec); i++)
        for (j = 0; j < 20000; j++);
}
#endif  MSDOS
```

BP.C

```
/* file: bp.c

     Do the actual work for the bp program.

     First version implemented by Elliot Jaffe


     Date of last revision:  8-12-87/JLM
     PIPELINE VERSION  13         Masscomp revisions
*/

#include "general.h"
#include "bp.h"
#include "variable.h"
#include "weights.h"
#include "patterns.h"
#include "command.h"

#define forkflag misc[0]
#define compout1flag misc[1]
#define comperrorflag1 misc[2]
#define comperrorflag2 misc[3]
#define compwedflag1 misc[4]
#define parentflag misc[5]
#define doneflag misc[6]
#define patno misc[7]
#define compout2flag misc[8]
#define compwedflag2 misc[9]
#define pss lumped[6*nunits]
#define tss lumped[6*nunits +1]
#define momentum lumped[6*nunits +2]
/*#define lrate lumped[6*nunits +3]*/

char   *Prompt = "bp: ";
char   *Default_step_string = "epoch";
char grain_string[20] = "pattern";
boolean System_Defined = FALSE;
boolean lflag = 1;
boolean cascade = 0;
int     epochno = 0;
```

```
int     cycleno = 0;
int     nepochs = 500;
int  ncycles = 50;
float   ecrit = 0.0;
float      crate = .05;
float      drate = .95;
float      gcor = 0.0;
int  follow = 0;
float  *netinput = NULL;
float  *activation = NULL;
float  *error = NULL;
float  *target = NULL;
float  *delta = NULL;
float  **dweight = NULL;
float  **pwed = NULL;
float  *dbias = NULL;
float  *pbed = NULL;
float   tmax = 1.0;
float      mu = .5;
int  tallflag = 0;


int    *misc = NULL;
extern float *lumped;
extern float **wtwed;
float *dwts = NULL;
float *wtss = NULL;
float *wds = NULL;

extern int read_weights();
extern int write_weights();

int  function, status, pid;
int miscid, miscpage;
unsigned miscsize;
int wtssid, wtsspage;
unsigned wtsssize;
int wdsid, wdspage;
unsigned wdssize;
int dwtsid,dwtspage;
unsigned dwtssize;
FILE *fopen(), *fp;

init_system() {
    int     strain (), ptrain (), tall (), test_pattern (),
reset_weights();
    int        get_unames(), set_lgrain(), cycle(), newstart();
    int change_lrate(), change_crate(), set_follow_mode();

    epsilon_menu = SETCONFMENU;

    init_weights();
```

```
    (void) install_command("strain", strain, BASEMENU,(int *) NULL);
    (void) install_command("ptrain", ptrain, BASEMENU,(int *) NULL);
    (void) install_command("tall", tall, BASEMENU,(int *) NULL);
    (void) install_command("test", test_pattern, BASEMENU,(int *)
NULL);
    (void) install_command("cycle", cycle, BASEMENU,(int *) NULL);
    (void)     install_command("reset",reset_weights,BASEMENU,(int
*)NULL);
    (void) install_command("newstart",newstart,BASEMENU,(int *)NULL);
    (void)  install_command("unames",  get_unames,  GETMENU,(int  *)
NULL);
    (void) install_command("patterns", get_pattern_pairs,
                                GETMENU,(int *) NULL);
    (void) install_var("lflag", Int,(int *) & lflag, 0, 0, SETPCMENU);
    (void) install_var("lgrain", String, (int *) grain_string,0,
0,NOMENU);
    (void) install_command("lgrain",set_lgrain,SETMODEMENU,(int *)
NULL);
    (void) install_var("follow", Int, (int *) & follow,0, 0,NOMENU);
    (void) install_command("follow",set_follow_mode,SETMODEMENU,(int
*)
NULL);
    (void) install_var("cascade", Int,(int *) & cascade, 0, 0,
SETMODEMENU);
    (void) install_var("nepochs", Int,(int *) & nepochs, 0, 0,
SETPCMENU);

    (void) install_var("ncycles", Int,(int *) & ncycles, 0, 0,
SETPCMENU);
    (void) install_var("epochno", Int,(int *) & epochno, 0, 0,
SETSVMENU);
/*    (void) install_var("patno", Int,(int *) & patno, 0, 0,
SETSVMENU); */
    (void) install_var("cycleno", Int,(int *) & cycleno, 0, 0,
SETSVMENU);
    init_pattern_pairs();
/*   (void) install_var("pss", Float,(int *) & pss, 0, 0, SETSVMENU);
    (void) install_var("tss", Float,(int *) & tss, 0, 0, SETSVMENU);*/
    (void) install_var("gcor", Float,(int *) & gcor, 0, 0, SETSVMENU);
/*    (void) install_var("momentum", Float,(int *)
&momentum,0,0,SETPARAMMENU);*/
    (void) install_var("mu", Float,(int *) &mu,0,0,SETPARAMMENU);
    (void) install_command("lrate", change_lrate, SETPARAMMENU, (int
*)
NULL);
    (void) install_command("crate", change_crate, SETPARAMMENU, (int
*)
NULL);
    (void) install_var("lrate", Float,(int *) & lrate, 0, 0, NOMENU);
    (void) install_var("crate", Float,(int *) & crate, 0, 0, NOMENU);
    (void) install_var("ecrit", Float,(int *) & ecrit, 0, 0,
SETPCMENU);
```

```
    (void) install_var("tmax", Float,(int *) & tmax, 0, 0,
SETPARAMMENU);
}

define_system() {
    register int    i,j, totnum;
    register float *wi, *shwi, *shdwt;
    register float *wt, *shwt, *wtend;
    float *tmp;

    if (!nunits) {
     put_error("cannot init bp system, nunits not defined");
     return(FALSE);
    }
    else
     if (!noutputs) {
         put_error("cannot init bp system, noutputs not defined");
         return(FALSE);
     }
    else
     if (!ninputs) {
         put_error("cannot init bp system, ninputs not defined");
         return(FALSE);
     }
    else
     if (!(nunits && noutputs && ninputs)) {
         put_error("cannot run bp system, nunits not defined");
         return(FALSE);
     }
netinput = &lumped[5*nunits];
    netinput = (float *) emalloc((unsigned)(sizeof(float) * nunits));

    (void) install_var("netinput", Vfloat,(int *) netinput, nunits, 0,

SETSVMENU);
    for (i = 0; i < nunits; i++)
     netinput[i] = 0.0;

    (void) install_var("pss", Float,(int *) &lumped[6*nunits], 0, 0,
SETSVMENU);
pss = 0.0;


    (void) install_var("tss", Float,(int *) &lumped[6*nunits + 1], 0,
0, SETSVMENU);
tss = 0.0;

    (void) install_var("momentum", Float,(int *) &lumped[6*nunits +
2],0,0,SETPARAMMENU);
momentum = 0.9;

/*   (void) install_var("lrate", Float,(int *) &lumped[6*nunits + 3],
```

```
0, 0, NOMENU);
lrate = 0.5;*/


/* activation, bias, target, error, delta arrays lumped here in shared

memory */
/* lumped[] defined in weights because of the order functions are
initialized*/

activation = &lumped[0];
/*    activation = (float *) emalloc((unsigned)(sizeof(float) *
nunits)); */
    (void) install_var("activation",Vfloat,(int
*)activation,nunits,0,SETSVMENU);
/*    for (i = 0; i < nunits; i++)
     activation[i] = 0.0;*/

delta = &lumped[nunits];
/* delta = (float *) emalloc((unsigned)(sizeof(float) * nunits));*/
(void) install_var("delta", Vfloat,(int *) delta, nunits, 0,
SETSVMENU);
/*    for (i = 0; i < nunits; i++)
     delta[i] = 0.0;*/

error = &lumped[2*nunits];
/*   error = (float *) emalloc((unsigned)(sizeof(float) * nunits));*/
    (void) install_var("error", Vfloat,(int *) error, nunits, 0,
SETSVMENU);
/*    for (i = 0; i < nunits; i++)
     error[i] = 0.0;*/

target = &lumped[7*nunits];
/*   target = (float *) emalloc((unsigned)(sizeof(float) *
noutputs));*/
  (void) install_var("target", Vfloat,(int *) target, noutputs,

          0, SETSVMENU);
/*    for (i = 0; i < noutputs; i++)
     target[i] = 0.0;*/

dweight = &wtwed[2*nunits];
/*    dweight = ((float **)
       emalloc((unsigned)(sizeof(float *)*nunits))); */
    (void) install_var("dweight", PVweight,(int *) dweight, nunits,
                            nunits, SETSVMENU);
totnum = 0;
for(i =0; i<nunits; i++)            /*get total number of inputs for

"s" variable of*/
     totnum += num_weights_to[i];   /* wed and weight arrays to
```

```
properly
size sh mem segment*/

/*dweight "s" array  defined  */

   dwtssize = (unsigned)(sizeof(float) *(nunits + totnum));
    dwtsid = shmget(0, dwtssize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("dwtsid is %10d\n", dwtsid);
errno = 0;
dwtspage = calc_page(dwtssize);
    dwts = (float *) shmat(dwtsid, dwtspage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

     for(i = 0; i< (nunits +totnum); i++)
     dwts[i] = 0.0;

/*    for (i = 0; i < nunits; i++) {
     dweight[i] = (float *)
          emalloc((unsigned)(sizeof(float)*num_weights_to[i]));
     for (j = 0; j < num_weights_to[i]; j++){
         dweight[i][j] = 0.0;
     }
    }
*/
dbias = &lumped[4*nunits];
/*    dbias = (float *) emalloc((unsigned)(sizeof(float) * nunits));
*/
    (void) install_var("dbias", Vfloat,(int *) dbias,
                              nunits, 0, SETSVMENU);
 /*    for (i = 0; i < nunits; i++)
     dbias[i] = 0.0; */

/* misc array for shmem flags */
errno = 0;
    miscsize = (unsigned)(sizeof(int) * 15);
    miscid = shmget(0, miscsize, 0666|IPC_CREAT);
if (errno >0)

fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("miscid is %10d\n", miscid);
errno = 0;
miscpage = calc_page(miscsize);
    misc = (int *) shmat(miscid, miscpage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
    (void) install_var("patno", Int,(int *) & misc[7], 0, 0,
```

```
SETSVMENU);

for (i = 0; i <12; i++)
     misc[i] = 0;
errno = 0;


/*weight "s" array  defined first */

    wtsssize = (unsigned)(sizeof(float) *(nunits +  totnum));
    wtssid = shmget(0, wtsssize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("wtssid is %10d\n", wtssid);
errno = 0;
wtsspage = calc_page(wtsssize);
    wtss = (float *) shmat(wtssid, wtsspage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

/* now wed  "s" array defined*/
    wdssize = (unsigned)(sizeof(float) *(nunits + totnum));
    wdsid = shmget(0, wdssize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("wdsid is %10d\n", wdsid);
errno = 0;
wdspage = calc_page(wdssize);
    wds = (float *) shmat(wdsid, wdspage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

/* now get contents of wed and weight "s" arrays and put them into
shemem wed and wts arrays */
shwt = &wtss[0];
shwi = &wds[0];
for(i = 0; i< nunits; i++){
     wt = weight[i];
     wi = wed[i];
     wtend = wt + num_weights_to[i];
     for(; wt < wtend ;){

     *shwt++ = *wt++;
     *shwi++ = *wi++;
        } /*end wt for*/
}  /*end for*/
```

```
/* now attach proper addresses to wed, weight and dweight "r" arrays
to
access repsective "s" arrays*/
totnum = 0;
for(i = 0; i <nunits; i++) {
     if(num_weights_to[i] !=0){
        weight[i] = &wtss[totnum];
        wed[i] = &wds[totnum];
           dweight[i] = &dwts[totnum];

        totnum += num_weights_to[i];
     } /* end if */
} /* end for */

    System_Defined = TRUE;
    return(TRUE);
}

float  logistic (x)
float  x;
{
    double  exp ();

#ifdef MSDOS
/* we are conservative under msdos to avoid potential underflow
   problems that may arise from returning extremal values -- jlm */
    if (x > 11.5
     return(.99999);
      else
     if (x < -11.5129)
         return(.00001);
#else
/* .99999988 is very close to the largest single precis value
   that is resolvably less than 1.0 -- jlm */
      if (x > 16.0)
     return(.99999988);
      else
     if (x < -16.0)
         return(.00000012);
#endif
    else
       return(1.0 / (1.0 + (float) exp( (double) ((-1.0) * x))));
}

init_output() {
    register int i,j;
    register float *sender, *wt, *end;
    float net;


    /* initializes the network to asymptotic outputs given 0 input */
    cycleno = 0;
```

```
    for (i = ninputs; i < nunits; i++) {/* to this unit */
     net = bias[i];
     sender = &activation[first_weight_to[i]];
     wt = weight[i];
     end = sender + num_weights_to[i];
     for (j = first_weight_to[i]; j < ninputs; j++) {
           sender++; wt++; /* step over input units to
                           initialize to all-zero input case */
     }
     for (; sender < end ; ){/* from this unit */
         net += (*sender++) * (*wt++);
     }
     netinput[i] = net;
     lumped[i] = activation[i] = (float) logistic(net);
    }
    if (step_size < PATTERN) {
     update_display();
     if (single_flag) {
         if (contin_test() == BREAK) return (BREAK);
     }
    }
    if (Interrupt) {
         Interrupt_flag = 0;
         update_display();
         if (contin_test() == BREAK) return (BREAK);
    }
    return(CONTINUE);
}

cycle() {
    register int i,cy;
    register float *sender,*wt,*end;
    float newinput;

    for (cy = 0; cy < ncycles; cy++) {
     cycleno++;
     for (i = ninputs; i < nunits; i++) {/* to this unit */
         newinput = bias[i];
         sender = &activation[first_weight_to[i]];
         end = sender + num_weights_to[i];
         wt = weight[i];
         for (;sender<end;) {/* from this unit */
           newinput += (*sender++) * (*wt++);
         }
         netinput[i] = crate * newinput + drate * netinput[i];
         activation[i] = (float) logistic(netinput[i]);
     }
     if (step_size == CYCLE) {
         update_display();
         if (single_flag) {
           if (contin_test() == BREAK) return (BREAK);
```

```
        }
      }
      if (Interrupt) {
          update_display();
          Interrupt_flag = 0;
          if (contin_test() == BREAK) return (BREAK);
      }
     }
     if (step_size == NCYCLES) {
      update_display();
     }
     return(CONTINUE);
}


compute_output() {
     register int    i;
     float *sender, *wt, *end;
     float net;
     compout2flag = 0;
     for (i = ninputs; i < nunits; i++) {  /* to this unit */
      net = bias[i];
      sender = &activation[first_weight_to[i]];
         end = sender + num_weights_to[i];
      wt = weight[i];
      for (; sender < end ;)
          net += (*sender++)*(*wt++); /* from this unit */
      netinput[i] = net;
activation[i] =(float) (1.0 / (1.0 + (float) exp( (double) ((-1.0) *
net))));
     }
      compout2flag = 1;
}

compute_error() {
     register int i,j;
     float *wt, *sender, *end;
     float del;
comperrorflag1 = 0;
comperrorflag2 = 0;

for (i = ninputs; i < nunits - noutputs; i++) {
      error[i] = 0.0;
}
while(!compout2flag);
compout2flag = 0;
for (i=nunits-noutputs, j=0; i < nunits ; j++, i++){
      if(target[j] >= 0)  /*  We care about this one*/
         error[i]= target[j] - activation[i];
      else
          error[i] = 0.0;
 }
```

```
comperrorflag1 = 1;
   for (i= nunits - 1; i >= ninputs; i--) {

     del   =   delta[i]   =   error[i]   *   activation[i]   *   (1.0   -
activation[i]);
     if (first_weight_to[i] + num_weights_to[i] < ninputs) continue;
     /* no point in propagating error back to input units */
     sender = &error[first_weight_to[i]];
     end = sender +    num_weights_to[i];
     wt = weight[i];
     for (;sender < end;) {
         *sender++ += del * (*wt++);
     }
    }
comperrorflag2 = 1;
}

compute_wed() {
    register int   i;
    float *wi, *sender, *end;
    float del;
    compwedflag2 = 0;
while (!comperrorflag2);
comperrorflag2 = 0;
   for (i = ninputs; i < nunits; i++) {
     sender = &activation[first_weight_to[i]];
     end = sender + num_weights_to[i];
     del = delta[i];
     wi = wed[i];
     for (;sender < end;)
         *wi++ += del * (*sender++);

     if( i <=(nunits - 1)) compwedflag1 = 1;
  }
compwedflag2 = 1;
}

clear_wed() {
    register int   i,j,num;
    register float *wi, *end;

    for (i = ninputs; i < nunits; i++) {
     bed[i] = 0.0;
     wi = wed[i];
     end = wi + num_weights_to[i];
     for (; wi < end;) {
         *wi++ = 0.0;
     }
    }
}

change_weights() {
```

```
    register int    i;
    register float *wt, *dwt, *epi, *wi, *end;

  for (i = ninputs; i < nunits; i++) {
while(!compwedflag1 && !compwedflag2);
     compwedflag1 = 0;

     dbias[i] = lrate*delta[i]+ momentum * dbias[i];
     /*lrate vs bepsilon delta[i]*/
     bias[i] += dbias[i];
     wt = weight[i];
     dwt= dweight[i];
     wi = wed[i];
     end = wt + num_weights_to[i];
     for (; wt < end; ) {
     *dwt = lrate * (*wi) + momentum * (*dwt); /*lrate vs (*epi++) */
         *wt++ += *dwt++;
         *wi++ = 0.0;
     }
    }
  /*  pos_neg_constraints();*/
}

float p_css = (float) 0.0;
float css = (float) 0.0;

change_weights_follow() {
    register int    i;
    register float *wt, *dwt, *epi, *wi, *end, *pwi;
    float tb, dp, den;

    p_css = css;
    css = 0.0;
    dp = 0.0;

    link_sum();

    for (i = ninputs; i < nunits; i++) {
        tb = bed[i];
     dbias[i] = tb*bepsilon[i] + momentum * dbias[i];
     bias[i] += dbias[i];
     css += ((double) tb)*((double) tb);
     dp += ((double) tb)*((double) pbed[i]);
     pbed[i] = tb;
     bed[i] = 0.0;
     wt = weight[i];
     dwt= dweight[i];
     wi = wed[i];
     pwi = pwed[i];
     epi = epsilon[i];
     end = wt + num_weights_to[i];
     for (; wt < end; ) {
```

```
            *dwt = (*epi++)*(*wi) + momentum * (*dwt);
            *wt++ += *dwt++;
            css += ((double) (*wi))*((double) (*wi));
            dp += ((double) (*wi))*((double) (*pwi));
            *pwi++ = *wi;
            *wi++ = 0.0;
        }
    }


    den = p_css * css;
    if (den > 0.0) gcor = dp/(sqrt(den));
    else gcor = 0.0;

    pos_neg_constraints();
}

constrain_weights() {
    pos_neg_constraints();
    link_constrain.s();
}

pos_neg_constraints() {
    float **fpt;

    for (fpt = positive_constraints; fpt && *fpt; fpt++)
     if (**fpt < 0.0)
         **fpt = 0.0;

    for (fpt = negative_constraints; fpt && *fpt; fpt++)
     if (**fpt > 0.0)
         **fpt = 0.0;
}

link_constraints() {
    register int    i,j;
    float   t;

    for (i = 0; i < nlinks; i++) {
     t = *constraints[i].cvec[0];
     for (j = 1; j < constraints[i].num; j++) {
         *constraints[i].cvec[j] = t;
     }
    }
}

link_sum() {
    register int    i,j;
    float   ss;

    for (i = 0; i < nlinks; i++) {
     ss = 0.0;
```

```
        for (j = 0; j < constraints[i].num; j++) {
            ss += *constraints[i].ivec[j];
        }
        for (j = 0; j < constraints[i].num; j++) {
            *constraints[i].ivec[j] = ss;
        }
       }
}

setinput() {
    register int    i,prev_index;
    register float  *pp;


    for (i = 0, pp = ipattern[patno]; i < ninputs; i++, pp++) {
        activation[i] =  *pp;
    }

    strcpy(cpname,pname[patno]);
}

settarget() {
    register int    i;
    register float *pp;

    for (i = 0, pp = tpattern[patno]; i < noutputs; i++, pp++) {
     target[i] = *pp;
     if (target[i] == 1.0) {
         target[i] = tmax;
     }
     else if(target[i] == 0.0) {
             target[i] = 1 - tmax;
     }
    }
}

setup_pattern() {
    setinput();
    settarget();
}
tallcompute_error() {
    register int i, j;
    float *wt, *sender, *end;
    float del;

for (i = ninputs; i < nunits - noutputs; i++) {
     error[i] = 0.0;
    }
for (i=nunits-noutputs, j = 0; i < nunits ; j++, i++){
     if(target[j] >= 0)  /* I care about this one*/
        error[i]= target[j] - activation[i];
     else
```

```
        error[i] = 0.0;
}
  for (i= nunits - 1; i >= ninputs; i--) {
    del =delta[i] = error[i]* activation[i] * (1.0 - activation[i]);
    if (first_weight_to[i] + num_weights_to[i] < ninputs) continue;
    /* no point in propagating error back to input units */
    sender = &error[first_weight_to[i]];
    end = sender +    num_weights_to[i];
    wt = weight[i];
    for (;sender < end;) {
        *sender++ += del * (*wt++);
    }
   }
}


talltrial() {

 setup_pattern();
    if (cascade) {
     if (init_output() == BREAK) return (BREAK);
     if (cycle() == BREAK) return (BREAK);
    }
    else  {
        compute_output();
        if (step_size < PATTERN) {
           update_display();
           if (single_flag) {
               if (contin_test() == BREAK) return(BREAK);
           } /* end single */
     } /* end stepsize */
} /* end else */
    tallcompute_error();
comperrorflag1 = 1;
   sumstats();
    return (CONTINUE);
} /* end of talltrial*/

trial() {

if (!forkflag){
     forkflag = 1;
     errno = 0;
     pid = fork();
     if (errno >0)
fprintf(fp, "Trouble with fork, errno =%3d:  %s\n", errno,
sys_errlist[errno]);

}
if(pid == 0){
errno = 0;
function = mpadvise(MPA_CPU_SET, 4);
```

```
if(errno > 0)
fprintf(stderr, "At 2nd proc call errno is %3d, which means %s\n",
errno, sys_errlist[errno]);
errno = 0;
forkagain:
if(doneflag){
     forkflag = 0;
     exit(0);
     }
compute_error();

change_weights();
while(!parentflag);
parentflag = 0;
goto forkagain;

} /*end of pid eq 0 */
if(pid != 0) return;

} /* end of train */

sumstats() {
    register i      i,j;
    register f      ;
    pss = 0.0;

while(!comperrorflag1);
comperrorflag1 = 0;
    for (j = 0,i =nunits - noutputs; i <nunits; i++,j++) {
      if (target[j] >= 0) {
           t = error[i];
     pss += t*t;
      }
    }
    tss += pss;
}
ptrain() {
  return(train('p'));
}

strain() {
  return(train('s'));
}

train(c) char c; {
    int     t,i,old,npat;
    char    *str;
    parentflag = 0;
    doneflag = 0;
    forkflag = 0;

    if (!System_Defined)
```

```
     if (!define_system())
         return(BREAK);

/*in case prev epoch was terminated early we clear the weds and beds
*/
    clear_wed();
    cycleno = 0;
    for (t = 0; t < nepochs; t++) {
      epochno++;
     for (i = 0; i < npatterns; i++)
         used[i] = i;
     if (c == 'p') {
       for (i = 0; i < npatterns; i++) {
         npat = rnd() * (npatterns - i) + i;
         old = used[i];
         used[i] = used[npat];
         used[npat] = old;
       }
     }
     tss = 0.0;

     for (i = 0; i < npatterns; i++) {

       patno = used[i];
     if(!forkflag){
     setinput();
     settarget();
        trial();
     } /* end of forkflag */

compute_output();
sumstats();
if(lflag) {
compute_wed();
     if (i != (npatterns -1)) {
           patno = used[i+1];
           setinput();
           settarget();
           parentflag = 1;
     } /* end of i not eq last pattern */

} /* end of lflag */

} /* end of npatterns for loop*/
if (tss < ecrit) break;
     if (t!= (nepochs - 1)){
     patno = used[0];
     setinput();
     settarget();
     parentflag = 1;
     } /* end of t ne loop */
```

```
        if (Interrupt) {
          Interrupt_flag = 0;
          update_display();
          if (contin_test() == BREAK) return(BREAK);
        }

     if (step_size == EPOCH) {   /* defined as 4*/
      update_display();
      if (single_flag) {
              if (contin_test() == BREAK) return(BREAK);
        } /*end of single flag*/
      } /* end of EPOCH */
}  /* end of nepochs for loop */

 doneflag = 1;
 parentflag = 1;
if (step_size == NEPOCHS) {   /* defined as 5 */
      update_display();
    }
    return(CONTINUE);
}

talltrain() {
    int     t,i,old,npat;
    char    *str;


    if (!System_Defined)
     if (!define_system())
         return(BREAK);
/*in case prev epoch was terminated early we clear the weds and beds
*/
       cycleno = 0;
     tss = 0.0;
     for (i = 0; i < npatterns; i++) {
       patno = used[i] = i;
           if(talltrial() == BREAK) return(BREAK);

         if (step_size == PATTERN) {
           update_display();
           if (single_flag) {
               if (contin_test() == BREAK) return(BREAK);
           }
         }
         if (Interrupt) {
           Interrupt_flag = 0;
           update_display();
           if (contin_test() == BREAK) return(BREAK);
         }
     } /* end of npatterns for loop*/

return(CONTINUE);
```

```
} /* end of talltrain */

tall() {
  int save_lflag;
  int save_single_flag;
  int save_nepochs;
  int save_step_size;

  save_lflag = lflag;  lflag = 0;
  save_single_flag = single_flag;
  if (in_stream == stdin) single_flag = 1;
  save_step_size = step_size;
  if (step_size > PATTERN) step_size = PATTERN;
  save_nepochs = nepochs;  nepochs = 1;
  tallflag = 1;
  talltrain();
  tallflag = 0;
  lflag = save_lflag;
  nepochs = save_nepochs;
  single_flag = save_single_flag;
  step_size = save_step_size;
  return(CONTINUE);
}

test_pattern() {
    char   *str;
    int save_single_flag;
    int save_step_size;


    if (!System_Defined)
     if (!define_system())
         return(BREAK);

    tss = 0.0;

    str = get_command("Test which pattern? ");
    if(str == NULL) return(CONTINUE);
    if ((patno = get_pattern_number(str)) < 0) {
       return(put_error("Invalid pattern specification."));
    }
    if (cascade) {
        save_single_flag = single_flag; single_flag = 1;
     save_step_size = step_size; step_size = CYCLE;
    }
    talltrial();
    update_display();
    if (cascade) {
     single_flag = save_single_flag;
     step_size = save_step_size;
    }
    return(CONTINUE);
```

```
}

newstart() {
     random_seed = rand();
     reset_weights();
}

reset_weights() {
    register int    i,j,k,first,num;
    char ch;

    epochno = 0;
    pss = tss = gcor = 0.0;
    cpname[0] = '\0';
    srand(random_seed);

    if (!System_Defined)
     if (!define_system())
         return(BREAK);

    for (j = 0, k = 3*nunits; j < nunits; j++, k++) {
     first = first_weight_to[j];
     num = num_weights_to[j];
      for (i = 0; i < num; i++) {
     wed[j][i] = dweight[j][i] = 0.0;
     if (pwed) pwed[j][i] = 0.0;
     ch = wchar[j][i];
     if (isupper(ch)) ch = tolower(ch);
     if (ch == '.') {
         weight[j][i] = 0.0;
     }
     else {

         if (constants[ch - 'a'].random) {
               if (constants[ch - 'a'].positive) {
                 weight[j][i] = wrange * rnd();
               }
               else
                 if (constants[ch - 'a'].negative) {
                     weight[j][i] = wrange * (rnd() - 1);
                 }
               else
                 weight[j][i] = wrange * (rnd() -.5);
         }
         else {
               weight[j][i] = constants[ch - 'a'].value;
         }
     }
      }
      bed[j] = dbias[j] = 0.0;
      if (pbed) pbed[j] = 0.0;
      ch = bchar[j];
```

```
        if (isupper(ch)) ch = tolower(ch);
        if (ch == '.') {
           bias[j] = 0;
        }
        else {
           if (constants[ch - 'a'].random) {
                 if (constants[ch - 'a'].positive) {
                   bias[j] = wrange * rnd();
                 }
                 else
                   if (constants[ch - 'a'].negative) {
                       bias[j] = wrange * (rnd() - 1);
                   }
                 else
                   bias[j]  = wrange * (rnd() -.5);
           }
           else {
                 bias[j]  = constants[ch - 'a'].value;
           }
        }
      }
      constrain_weights();
      for (i = 0, j = 4*nunits; i < noutputs; i++, j++) {
        lumped[j] = target[i] = 0.0;
      }
      for (i = 0; i < nunits; i++)
        netinput[i] = activation[i] = delta[i] = error[i] = 0.0;

      for (i = 0; i < 3*nunits; i++)
       lumped[i] = 0.0;

      update_display();
      return(CONTINUE);
}


set_lgrain() {
      char old_grain_string[STRINGLENGTH];
      struct Variable *vp, *lookup_var();

      strcpy(old_grain_string,grain_string);

      vp = lookup_var("lgrain");
      change_variable("lgrain",vp);

      if(startsame(grain_string,"epoch"))strcpy(grain_string,"epoch");
      else if (startsame(grain_string,"pattern"))
                              strcpy(grain_string,"pattern");
      else {
       strcpy(grain_string,old_grain_string);
       return(put_error("urecognized grain -- not changed."));
      }
```

```
    return(CONTINUE);
}

set_follow_mode() {
    struct Variable *vp, *lookup_var();
    int pv, i, j;
    pv = follow;

    vp = lookup_var("follow");
    change_variable("follow",vp);

    if (follow == 0) return (CONTINUE);
    if (pwed == NULL) {
      pwed = ((float **) emalloc((unsigned)(sizeof(float *)*nunits)));
      (void) install_var("pwed", PVweight,(int *) pwed, nunits,
                              nunits, NOMENU);
      for (i = 0; i < nunits; i++) {
     pwed[i] = ((float *)
           emalloc((unsigned)(sizeof(float)*num_weights_to[i])));
      }

      pbed = ((float *) emalloc((unsigned)(sizeof(float) * nunits)));
      (void) install_var("pbed", Vfloat,(int *) pbed,
                              nunits, 0, NOMENU);
    }
    if (pv == 0) {
      for (i = 0; i < nunits; i++) {
     for (j = 0; j < num_weights_to[i]; j++) {
         pwed[i][j] = 0.0;
     }
      }
      for (i = 0; i < nunits; i++)
     pbed[i] = 0.0;
    }
    gcor = css = 0.0;
    return(CONTINUE);
}


change_crate() {
    struct Variable *varp;

    if ((varp = lookup_var("crate")) != NULL) {
     change_variable("crate",(int *) varp);
    }
    else {
     return(put_error("crate is not defined"));
    }
    drate = 1 - crate;
    return(CONTINUE);
}
```

```
init_weights() {
    int define_bp_network();
    (void) install_command("network", define_bp_network,GETMENU,(int
*)
NULL);
    (void) install_command("weights", read_weights, GETMENU,(int *)
NULL);
    (void) install_command("weights", write_weights, SAVEMENU,(int *)
NULL);
    (void) install_var("nunits", Int,(int *) & nunits, 0, 0,
SETCONFMENU);
    (void) install_var("ninputs", Int,(int *) & ninputs, 0, 0,
SETCONFMENU);
    (void) install_var("noutputs", Int,(int *) & noutputs, 0, 0,
SETCONFMENU);
    (void) install_var("wrange",Float,(int *) &wrange,0,0,
SETPARAMMENU);
}
```

WEIGHTS.C

```
/* file: weights.c

     read in network descriptions, and set up constraints.
     First version implemented by Elliot Jaffe.
     Date of last revision: 8-12-87/JLM.

     Masscomp revisions pipeline ver 13  */

/*LINTLIBRARY*/

/* the following is the form for network description files.

definitions:
nunits <int>
ninputs <int>
noutputs <int>
maxconstraints <int>
constraints:
<char> <float> or <char> [random positive negative linked]
...
end

network:
<strings of . and chars as defined in definitions:>
end
biases:
<a single line of . and chars as biases for units>
end
sigmas:
<a single line of .'s and chars specifying sigmas -- harmony theory
only>
```

```
end
<EOF>
*/


#include "general.h"
#include "command.h"
#include "weights.h"
#include "variable.h"

float **weight = NULL;
char   **wchar;              /* pointers to vectors of chars
                                that are used in resetting weights*/
float  *bias = NULL;
char   *bchar;                 /* like wchar */
float **epsilon;
float  *bepsilon = NULL;     /* thresh epsilon array */
float  **wed = NULL;
float  *bed = NULL;
float  *sigma = NULL;  /* strength parameter for knowledge atoms */

struct constants    constants[26];

float **positive constraints;
float **nega        traints;
 /* Array of s.    nstraint, for keeping links together */
struct constraint  *constraints = NULL;

float   lrate = 0.5;

float   wrange = 1;

int     nunits = 0;
int     ninputs = 0;
int     noutputs = 0;
int  maxpos = MAXCONSTRAINTS;
int  maxneg = MAXCONSTRAINTS;
int     nlinks = 0;
static  nposconstr = 0;
static  nnegconstr = 0;
int     epsilon_menu = SETWTMENU;
char    net_descr_name[BUFSIZ];

int lumpid, lumppage;  /* added for shared mem 11 apr 89 bb */
unsigned lumpsize;

float *lumped = NULL;

int fnwtid, fnwtpage;
unsigned fnwtsize;
int *fnwt = NULL;
```

```
int wtwedid, wtwedpage;
unsigned wtwedsize;
float **wtwed = NULL;

int bp;    /* TRUE if program is bp */

# define ENLARGE_POS -1
# define ENLARGE_NEG -2

define_bp_network() {
     bp = 1;
     define_net();
}

define_network() {
     bp = 0;
     define_net();
}

define_net() {
    char   *sp;
    char    string[BUFSIZ];
    FILE * sv_instream;
    struct Variable *lookup_var ();
    int     i;
    boolean defined_weights = FALSE;

    sv_instream = in_stream;
    sp = get_command("filename for network description: ");
    if ( sp == NULL) return(CONTINUE);
    strcpy(net_descr_name,sp);

    if ((in_stream = fopen(sp, "r")) == NULL) {
     in_stream = sv_instream;
     return(put_error("Can't open network file."));
    }

    nlinks = 0;

    for (i = 0; i < 26; i++) {
     constants[i].random = FALSE;
     constants[i].positive = FALSE;
     constants[i].negative = FALSE;
     constants[i].link = FALSE;
     constants[i].value = 0.0;
    }

    constants['r' - 'a'].random = TRUE;

    constants['p' - 'a'].random = TRUE;
    constants['p' - 'a'].positive = TRUE;
    constants['n' - 'a'].random = TRUE;
```

```
constants['n' - 'a'].negative = TRUE;

while (fscanf(in_stream, "%s", string) != EOF) {
 if (!strcmp(string, "definitions:")) {
     if (read_definitions() == BREAK) {
         fclose(in_stream); in_stream = sv_instream;
       return(BREAK);
     }
 }
 else
     if (!strcmp(string, "constraints:")) {
       if (read_constraints(constants) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
       }
     }
 else
     if (!strcmp(string, "network:")) {
       defined_weights = read_network(constants);
       if (!defined_weights) {
        if (put_error(err_string) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
        }
       }
     }
 else
     if (!strcmp(string, "biases:")) {
       if (read_biases(constants) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
       }
     }
 else
     if (!strcmp(string, "sigmas:")) {
       if (read_sigmas(constants) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
       }
     }
 else
     if (!strcmp(string, "end")) {
     /* just skip over it */
     }
 else {
     sprintf(err_string,
  "error reading network file: I don't understand %s\n",string);
     if (put_error(err_string) == BREAK) {
       fclose(in_stream); in_stream = sv_instream;
       return(BREAK);
     }
```

```
        }
    }
    fclose(in_stream);
    in_stream = sv_instream;
    if (nlinks)
     constrain_weights();
    return(CONTINUE);
}

read_definitions() {
    char    string[BUFSIZ];
    struct Variable *varp,
                    *lookup_var ();

    while (fscanf(in_stream, "%s", string) != EOF) {
     if (!strcmp(string, "end"))
         return(CONTINUE);
     if ((varp = lookup_var(string)) != NULL) {
         change_variable(string,(int *) varp);
     }
     else {
         sprintf(err_string,
           "Error: unknown variable in network file, %s\n", string);
         return(put_error(err_string));
     }
    }
}

read_network(con)
struct constants   *con;
{
    int     i,r,s,block,since_first,last_weight_to,tempint;
    int        rstart,rnum,rend,sstart,snum,send,con_index;
    char    ch,all_ch,*strp;
    char    string[BUFSIZ];
    int        needline = 1;
    float   *tmp; char *ctmp;

    (void) srand(random_seed);

/* activation, bias, target, error, delta arrays lumped here in shared

memory */
errno = 0;
    lumpsize = (unsigned)(sizeof(float) *(7*nunits + noutputs));
    lumpid = shmget(0, lumpsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("lumpid is %10d\n", lumpid);
errno = 0;
lumppage = calc_page(lumpsize);
```

```
    lumped = (float *) shmat(lumpid, lumppage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);

for (i = 0; i <6*nunits + noutputs; i++)
     lumped[i] = 0.0;
errno = 0;

/* weight and wed "r" arrays for shared memory  */
    wtwedsize = (unsigned)(sizeof(float *) *(3*nunits));
    wtwedid = shmget(0, wtwedsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("wtwedid is %10d\n", wtwedid);
errno = 0;
wtwedpage = calc_page(wtwedsize);
wtwed = (float **) shmat(wtwedid, wtwedpage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

weight = &wtwed[0];
/*   weight      oat**) emalloc((unsigned int)(sizeof(float *) *
nunits))); */

    epsilon = ((float **) emalloc((unsigned int)(sizeof(float *)) *
nunits));

    wchar = ((char **) emalloc((unsigned int)(sizeof(char *) *
nunits)));

fnwtsize = (unsigned)(sizeof(int *) *(2*nunits));
    fnwtid = shmget(0, fnwtsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("fnwtid is %10d\n", fnwtid);
errno = 0;
fnwtpage = calc_page(fnwtsize);
fnwt = (int *) shmat(fnwtid, fnwtpage, 0);

if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

first_weight_to = &fnwt[0];
/*first_weight_to = (int *) emalloc((unsigned int)(sizeof(int) *
nunits)); */
/*    for (r = 0; r < nunits; r++)
     first_weight_to[r] = nunits;
*/
```

```
num_weights_to = &fnwt[nunits];
 /*num_weights_to = (int *) emalloc((unsigned int)(sizeof(int) *
nunits)); */

    for (r = 0; r < nunits; r++){
     num_weights_to[r] = 0;

        first_weight_to[r] = nunits;
    }

   (void) install_var("weight",PVweight,(int *) weight,nunits,nunits,
                                        SETWTMENU);
(void)  install_var("epsilon",  PVweight,(int  *)  epsilon,  nunits,
nunits,
                                        epsilon_menu);
    if (bp) {
wed = &wtwed[nunits];
/*     wed = ((float**) emalloc((unsigned int)(sizeof(float *) *
nunits))); */
     (void) install_var("wed",PVweight,(int *) wed,nunits,nunits,
                                        SETSVMENU);
    }

    rstart = 0; rend = nunits -1; sstart = 0; send = nunits -1;
    for (block = 0; ; block++) {
gbagain:
      if (fscanf(in_stream,"%s",string) == EOF) {
         sprintf(err_string,"error in network description");
         return(FALSE);
      }
      if (strcmp("end",string) == 0) {
           if (block) return(TRUE);
     else {
      sprintf(err_string,"error in network description");
     }
     return(FALSE);
      }
      all_ch = '\0';
      if (string[0] == '%') {
        fscanf(in_stream,"%d%d%d%d",&rstart,&rnum,&sstart,&snum);
     rend = rstart + rnum -1;
     send = sstart + snum -1;
     if (string[1]) {
         all_ch = string[1];
     }
      }
      else {
       if (!block) {
     needline = 0;
       }
       else {
         sprintf(err_string,"error in network description");
```

```
        return(FALSE);
       }
      }
      for (r = rstart; r <= rend; r++) {
       if (!all_ch) {
         if (needline) {
             if (fscanf(in_stream,"%s",string) == EOF) {
            sprintf(err_string,"not enough units in network
description");
            return(FALSE);

          }
         }
         else needline = 1;
       }
       else {
         for (s = 0; s < snum; s++) string[s] = all_ch;
         string[s] = '\0';
       }
       first_weight_to[r] = sstart;
       last_weight_to = send;
       num_weights_to[r] = 1 + last_weight_to - first_weight_to[r];
  weight[r] = ((float *)
         emalloc ((unsigned int)(sizeof(float) * num_weights_to[r])));

  epsilon[r] = ((float *)
            emalloc    ((unsigned    int)(sizeof(float)    *
num_weights_to[r])));

  wchar[r] = ((char *)
             emalloc    ((unsigned    int)(sizeof(char)    *
num_weights_to[r])));
       if (bp) {
  wed[r] = ((float *)
     emalloc ((unsigned int)(sizeof(float) * num_weights_to[r])));

       }
       for(s = 0; s < num_weights_to[r]; s++) {
          weight[r][s] = 0.0;
          epsilon[r][s] = 0.0;
          wchar[r][s] = '.';
          if (bp) wed[r][s] = 0.0;
       }
       for (strp = string,s = sstart,since_first = 0; s <= send; s++)
{
                        /* loop over the from units */
         ch = *strp++;
         wchar[r][since_first] = ch;
         if (ch == '.') {
           since_first++;
         }
         else {
```

```
        /* first check if this is realy a character */
          if (!isalpha(ch)) {
              sprintf(err_string,"non_alpha character in network");
              return(FALSE);
          }

        /* upper case means this weight is non-changable */
          if (isupper(ch)) {
          /* make it lower case */
              ch = tolower(ch);
              epsilon[r][since_first] = 0;
          }
          else {
              epsilon[r][since_first] = lrate;
          }


        /* now set up the char based on the stored con definitions */
          if (con[ch - 'a'].random) {
              if (con[ch - 'a'].positive) {
                if (nposconstr >= maxpos) {
                    enlarge_constraints(ENLARGE_POS);
                }
                weight[r][since_first] = wrange * rnd();
                positive_constraints[nposconstr++] =
                     &weight[r][since_first];
              }
              else
                if (con[ch - 'a'].negative) {
                    if (nnegconstr >= maxneg){
                      enlarge_constraints(ENLARGE_NEG);
                    }
                    weight[r][since_first] =
                         wrange * (rnd() - 1);
                    negative_constraints[nnegconstr++] =
                         &weight[r][since_first];
                }
              else
                weight[r][since_first] = wrange * (rnd() -.5);
          }
          else {
              weight[r][since_first] = con[ch - 'a'].value;
          }
          if (con[ch - 'a'].link) {
              con_index = (con[ch - 'a'].link - 1);
              if (constraints[con_index].num >=
constraints[con_index].max) {
                    enlarge_constraints(con_index);
              }

              tempint = constraints[con_index].num;
              constraints[con_index].cvec[tempint]
```

```
                    = &weight[r][since_first];

                if (bp) {
                  constraints[con_index].ivec[tempint]
                      = &wed[r][since_first];
                }

                  tempint = constraints[con_index].num + 1;
                constraints[con_index].num = tempint;
                /* this kludge (tempint) is for the MS compiler */
              }
            since_first++;
            }
          }
        }
      }
}

read_biases(con)

struct constants   *con;
{
    int     j,rstart,rend,rnum,block,con_index,tempint;
    char    ch,all_ch,*strp;
    char    string[BUFSIZ];

bias = &lumped[3*nunits];
/*bias = (float *) emalloc((unsigned int)(sizeof(float) * nunits));*/
    (void) install_var("bias", Vfloat,(int *) bias, nunits, 0,
SETWTMENU);

    bepsilon = (float *) emalloc((unsigned int)(sizeof(float) *
nunits));
    (void) install_var("bepsilon", Vfloat,(int *) bepsilon, nunits, 0,

                                          epsilon_menu);
    bchar = (char *) emalloc((unsigned int)(sizeof(char) * nunits));

    if (bp) {
     bed = (float *) emalloc((unsigned int)(sizeof(float) * nunits));
        (void) install_var("bed", Vfloat,(int *)
bed,nunits,0,SETSVMENU);
    }

    for (j = 0; j < nunits; j++) {
     bias[j] = 0.0;
     bepsilon[j] = 0;
     bchar[j] = '.';
     if (bp) bed[j] = 0.0;
    }

    rstart = 0; rend = nunits -1;
```

```
  for (block = 0; ; block++) {
gtagain:
    if (fscanf(in_stream,"%s",string) == EOF) {
     return(put_error("problem in bias description"));
    }
    if (strcmp(string,"end") == 0) {
        if (block) return (CONTINUE);
        else return(put_error("problem in bias description"));
    }
    if (string[0] == '%') {
        fscanf(in_stream,"%d%d",&rstart,&rnum);
     rend = rstart + rnum -1;
     if (string[1] != '\0') {
         all_ch = string[1];
         for (j = 0; j < rnum; j++) {
           string[j] = all_ch;
         }
         string[j] = '\0';
     }
     else goto gtagain;
    }
    for (strp = string, j = rstart; j <= rend; j++, strp++) {
     ch = *strp;

     bchar[j] = ch;
     if (ch == '.') {
         bias[j] = 0;
         bepsilon[j] = 0;
     }
     else {
     /* first check if this is realy a character */
         if (!isalpha(ch)) {
           return(put_error("non_alpha character in bias"));
         }

     /* upper case means this weight is non-changable */
         if (isupper(ch)) {
         /* make it lower case */
           ch = tolower(ch);
           bepsilon[j] = 0;
         }
         else {
           bepsilon[j] = lrate;
         }

         /* now set up the char based on the stored con definitions */
         if (con[ch - 'a'].random) {
               if (con[ch - 'a'].positive) {
                 bias[j] = wrange * rnd();
                 if (nposconstr >= maxpos) {
                     enlarge_constraints(ENLARGE_POS);
                 }
```

```
                positive_constraints[nposconstr++] = &bias[j];
              }
              else
                if (con[ch - 'a'].negative) {
                    bias[j] = wrange * (rnd() - 1);
                    if (nnegconstr >= maxneg){
                      enlarge_constraints(ENLARGE_NEG);
                    }
                    negative_constraints[nnegconstr++] = &bias[j];
                }
              else
                bias[j] = wrange * (rnd() -.5);
          }
          else {
                bias[j] = con[ch - 'a'].value;
          }
          if (con[ch - 'a'].link) {
            con_index = (con[ch - 'a'].link - 1);
            if (constraints[con_index].num >=
constraints[con_index].max){
                enlarge_constraints(con_index);
              }
            tempint = constraints[con_index].num;
            constraints[con_index].cvec[tempint] = &bias[j];
            if (bp) constraints[con_index].ivec[tempint] = &bed[j];
            constraints[con_index].num++;

          }
      }
    }
  }
}

read_sigmas(con) struct constants   *con; {
    int     j;
    char    ch, all_ch, *strp;
    char    string[BUFSIZ];
    int        rstart, rend, rnum, block;

    sigma = (float *) emalloc((unsigned int)(sizeof(float) * nunits));
    for (j = 0; j < nunits; j++) {
      sigma[j] = 1.0;        /* default sigma is 1.0 */
    }
    (void) install_var("sigma", Vfloat,(int *) sigma, nunits, 0,
                  SETWTMENU);
    rstart = 0; rend = nunits -1;
  for (block = 0; ; block++) {
gsagain:
    if (fscanf(in_stream, "%s", string) == EOF) {
     return(put_error("problem in sigma description"));
    }
    if (strcmp(string,"end") == 0) {
```

```
            if (block) return (CONTINUE);
            else return(put_error("problem in sigma description"));
        }
        if (string[0] == '%') {
            fscanf(in_stream,"%d%d",&rstart,&rnum);
         rend = rstart + rnum -1;
         if (string[1] != '\0') {
             all_ch = string[1];
             for (j = 0; j < rnum; j++) {
               string[j] = all_ch;
             }
             string[j] = '\0';
         }
         else goto gsagain;
        }
        for (strp = string, j = rstart; j <= rend; j++, strp++) {
         ch = *strp;
         if (ch == '.') {
           sigma[j] = 1.0;
         }
         else {
         /* first check if this is really a character */
             if (!isalpha(ch)) {
               return(put_error("non_alpha character in bias"));
             }
             if (isupper(ch)) {
             /* make it lower case */
               ch = tolower(ch);
             }

             sigma[j] = con[ch - 'a'].value;
             if (sigma[j] < 0) {
               return(put_error("can't set sigma less than 0!"));
             }
         }
        }
      }
}

read_constraints(con)
struct constants   *con;
{
    char    ch;
    float   flt;
    int     isflt;
    char    string[BUFSIZ];
    char    str[5][30];
    int     i,j,ch_ind;
    int        nstr;

    while (fgets(string, BUFSIZ, in_stream) != NULL) {
     if (string[0] == NULL || string[0] == '\n') {
```

```
            if (fgets(string, BUFSIZ, in_stream) == NULL) {
              break;
            }
        }
        if (strncmp(string,"end",3) == 0) break;

        ch = '\0';

        for (i = 0; i < 5; i++) str[i][0] = '\0';

        (void) sscanf(string, "%c %s %s %s %s %s",
                      &ch, str[0], str[1], str[2], str[3], str[4]);
        ch = (isupper(ch)) ? tolower(ch) : ch;
        ch_ind = ch - 'a';
        con[ch_ind].random = con[ch_ind].positive =
            con[ch_ind].negative = con[ch_ind].link = FALSE;
        con[ch_ind].value = 0.0;
        for (i = 0; (i < 5) && (str[i][0] != '\0'); i++) {
            if ( (isflt = sscanf(str[i],"%f",&flt)) == 1) {
              con[ch_ind].value = flt;
            }
            else
              if (startsame(str[i], "random"))
                  con[ch_ind].random = TRUE;
            else
              if (startsame(str[i], "positive"))
                  con[ch_ind].positive = TRUE;
            else
              if (startsame(str[i], "negative"))
                  con[ch_ind].negative = TRUE;
            else
              if (startsame(str[i], "linked"))

                  con[ch_ind].link = ++nlinks;
            else {
              sprintf(err_string,
                "unknown type for constant %c, %s\n", ch, str[i]);
              if (put_error(err_string) == BREAK) {
                    return(BREAK);
              }
            }
        }
       }
       if (nlinks) {
        constraints = (struct constraint   *)
          emalloc ((unsigned int)(sizeof (struct constraint) * (nlinks +
1)));
        for (i = 0; i < nlinks; i++) {
            constraints[i].num = 0;
            constraints[i].max = MAXCONSTRAINTS;
            constraints[i].cvec = ((float **)
                emalloc((unsigned int)(sizeof(float *) *
```

```
MAXCONSTRAINTS)));
        constraints[i].ivec = ((float **)
            emalloc((unsigned int)(sizeof(float *) *
MAXCONSTRAINTS)));
        for (j = 0; j < nunits; j++) {
          constraints[i].cvec[j] = NULL;
          constraints[i].ivec[j] = NULL;
        }
     }
    }
    else {
     constraints = NULL;
    }
    positive_constraints = ((float **)
          emalloc((unsigned int)(sizeof(float *) * MAXCONSTRAINTS)));
    for (i = 0; i < MAXCONSTRAINTS; i++)
     positive_constraints[i] = NULL;
    negative_constraints = ((float **)
          emalloc((unsigned int)(sizeof(float *) * MAXCONSTRAINTS)));
    for (i = 0; i < MAXCONSTRAINTS; i++)
     negative_constraints[i] = NULL;
    return(CONTINUE);
}

change_lrate() {
    struct Variable *varp;
    int     i,
            j;

    if ((varp = lookup_var("lrate")) != NULL) {
     change_variable("lrate",(int *) varp);
    }
    else {
     return(put_error("BIG PROBLEM: lrate is not defined"));
    }


    if (epsilon != NULL) {
     for (i = 0; i < nunits; i++) {
        for (j = 0; j < num_weights_to[i]; j++) {
          if (epsilon[i][j] != 0.0)
              epsilon[i][j] = lrate;
        }
     }
    }
    if (bepsilon != NULL) {
     for (i = 0; i < nunits; i++) {
        if (bepsilon[i] != 0.0)
          bepsilon[i] = lrate;
     }
    }
}
```

```
/* given a defined system, we will write the matrix and the biases
out to a file. The file format is one floating point number per line,
with the weight matrix in row major format followed by the biases.
*/

write_weights() {
    int     i,j,end;
    char   *str = NULL;
    char fname[BUFSIZ];
    char *star_ptr;
    char tstr[40];
    FILE * iop;

    if (weight == NULL) {
     return(put_error("cannot save undefined network"));
    }

nameagain:
    str = get_command("weight file name: ");
    if (str == NULL) return(CONTINUE);
    strcpy(fname,str);
    if ( (star_ptr = index(fname,'*')) != NULL) {
     strcpy(tstr,star_ptr+1);
     sprintf(star_ptr,"%d",epochno);
     strcat(fname,tstr);
    }
    if ((iop = fopen(fname, "r")) != NULL) {
     fclose(iop);
        get_command("file exists -- clobber? ");
     if (str == NULL || str[0] != 'y') {
        goto nameagain;
     }
    }
    if ((iop = fopen(fname, "w")) == NULL) {
     return(put_error("cannot open file for output"));
    }


    for (i = 0; i < nunits; i++) {
       for (j = 0; j < num_weights_to[i]; j++) {
     fprintf(iop, "%f\n", weight[i][j]);
       }
    }

    if (bias) {
      for (i = 0; i < nunits; i++) {
      fprintf(iop, "%f\n", bias[i]);
      }
    }

    if (sigma) {
      for (i = 0; i < nunits; i++) {
```

```
      fprintf(iop, "%f\n", sigma[i]);
      }
    }

    (void) fclose(iop);
    return(CONTINUE);
}


read_weights() {
    int     i,j,end;
    register float *wt, *shwt, *wtend, *shend;
    char   *str = NULL;
    FILE * iop;
    if(!System_Defined)
       if(!define_system())
       return(BREAK);

    if (weight == NULL) {
     return(put_error("cannot restore undefined network"));
    }

    if((str = get_command("File name for stored weights: ")) == NULL)
     return(CONTINUE);

    if ((iop = fopen(str, "r")) == NULL) {
     sprintf(err_string,"Cannot open weight file %s.",str);
     return(put_error(err_string));
    }

    for (i = 0; i < nunits; i++) {
     if(num_weights_to[i] == 0) continue;
     for (j = 0; j < num_weights_to[i]; j++) {
       if (fscanf(iop, "%f", &weight[i][j]) == 0) {
       fclose(iop);
         return(put_error("weight file is not correct for this
network"));
       }
     }
    }


    end = nunits;

    if (bias != NULL) {
      for (i = 0; i < end; i++) {
     if (fscanf(iop, "%f", &bias[i]) == 0) {
         fclose(iop);
         return(put_error("weight file is not correct for this
network"));
     }
      }
```

```
    }

    if (sigma != NULL) {
      for (i = 0; i < end; i++) {
     if (fscanf(iop, "%f", &sigma[i]) == 0) {
         fclose(iop);
         return(put_error("weight file is not correct for this
network"));
     }
      }
    }
    (void) fclose(iop);
    update_display();
    return(CONTINUE);
}

/* realloc positive_constraints, negative_constraints, and link
constraints
   this is called whenever the allocated constraint lists run out of
   space for additional constraints  14-May-87 MAF / 15-May-87 JLM */

enlarge_constraints(con_index) int con_index; {
    if (con_index = ENLARGE_POS) {
     maxpos += 100;
     positive_constraints = ((float **) erealloc
       ((char *) positive_constraints,
           (unsigned int) ((maxpos - 100) * sizeof(float *)),
           (unsigned int) (maxpos * sizeof(float *))));
    }
    else if (con_index = ENLARGE_NEG) {
     maxneg += 100;
     negative_constraints = ((float **) erealloc
       ((char *) negative_constraints,
          (unsigned int) ((maxneg -100) * sizeof (float *)),
          (unsigned int) (maxneg * sizeof(float *))));
    }
    else {
     constraints[con_index].max += 100;
        constraints[con_index].cvec = ((float **) erealloc
       ((char *)constraints[con_index].cvec,
        (unsigned int)
          ((constraints[con_index].max - 100) * sizeof(float *)),
        (unsigned int)
          (constraints[con_index].max * sizeof(float *))));
        constraints[con_index].ivec = ((float **) erealloc
       ((char *)constraints[con_index].ivec,
        (unsigned int)
          ((constraints[con_index].max - 100) * sizeof(float *)),
        (unsigned int)
          (constraints[con_index].max * sizeof(float *))));
   }
}
```

Appendix 4

Modified SDMO Source Code for Pipeline Method

GENERAL.C

```
/*contains several general routines needed in the shared memory
version of the simulator*/

#include <stdio.h>
#include "main_def.h"

extern double *lumped;
extern int lumpid;
extern int *misc;
extern int miscid;

calc_page(size)
int size;
{
int inc, pg, endof, xtra, nuend;
inc = 1;
pg = 4096;
endof = sbrk(0);
while (pg <= endof) {
     pg = 4096 * inc;
     ++inc;
}
xtra = (pg - endof) + size;
nuend = sbrk(xtra);
return(pg);

} /* end of calc_page */

quit()
{

shmdt(lumped);
shmctl(lumpid, IPC_RMID, 0);
shmdt(misc);
shmctl(miscid, IPC_RMID, 0);
}/*  end of quit */
```

MAIN.C

```
/*-----------------------------------------------------------------*/
/*   Module Name: Main.c                                           */
/*   ===============                                               */
/*    This is the main body of the program.                        */
/*   Here a  neural network with unlimited number of inputs/outputs
```

```
  default:    /* actually the last layer */
          temp = on_this_network->no_of_layers - 1;
          for ( counter =0;
                counter < on_this_network->this_layer[temp]-
>no_of_neurons ; ++counter)
          {
              on_this_network->net_output_list[counter] =
                  on_this_network->this_layer[temp]-
>this_neuron[counter] ->output[0];
          }

          break;
  }
}  /* end of SEND_OUTER_LAYER  */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F
.10----------------------------------*/

process_neuron(this_neuron)
NEURON    *this_neuron;

/* Will just process the inputs add them according to law and
   store the output so the layer is an independent block  */
{  /* begin of PROCESS_NEURON  */
    int  counter;
    double   pow(), hold_value;

hold_value = this_neuron->bias[0];
for ( counter =0; counter < this_neuron->no_of_inputs ; ++counter)
    {
      hold_value += (this_neuron->weight_list[counter]) *
                            (this_neuron->input_list[counter]);
     }
 /* Here the actual function will be include as a pointer to have any
      transfer function available instead of the sharp threshold */
this_neuron->output[0] = sigmoid(hold_value, this_neuron->threshold );

}  /* end of PROCESS_NEURON  */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 1 1
---------------------------------*/

process_layer(this_layer)
LAYER   *this_layer;

{  /* begin of PROCESS_LAYER  */
    int counter;
    for( counter =0; counter < this_layer->no_of_neurons ; ++counter)
       process_neuron(this_layer->this_neuron[counter]);

}  /* end of PROCESS_LAYER  */
```

```
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 1 2
----------------------------------*/

process_network(this_network)
NETWORK   *this_network;

{  /* begin of PROCESS_NETWORK  */
    int counter;

  /* start with first layer...*/
   send_outer_layer(this_network, 0);
   process_layer(this_network->this_layer[0]);
   print_layer(this_network, 0);
   if ( this_network->no_of_layers > 1)
     send_layer(this_network, 0);

   /* Now go the inner(hidden layers...if any */
   if ( this_network->no_of_layers > 1)
for (counter=1; counter< (this_network->no_of_layers-1); ++counter){
       process_layer(this_network->this_layer[counter]);
       print_layer(this_network, counter);
       send_layer(this_network, counter);
     }


  /* Finally the last layer and done ...and handle for one layer only.

*/
  if ( this_network->no_of_layers > 1)
process_layer(this_network->this_layer[this_network->no_of_layers
-1]);
  send_outer_layer(this_network, 1  ); /* send outer anyway */
  if ( this_network->no_of_layers > 1)
    print_layer(this_network, this_network->no_of_layers -1);

}  /* end of PROCESS_NETWORK  */


process_network_b(this_network)
NETWORK   *this_network;
/* Same as previous routine..but it'll do it silently and will not
change the
   net_output/input_list vector on network ..used by backpropagation
train..*/

{  /* begin of PROCESS_NETWORK_B  */
    int counter;

  /* start with first layer...*/
   send_outer_layer(this_network, 0);
   process_layer(this_network->this_layer[0]);
   /* Now go the inner(hidden layers...if any */
```

```
   if ( this_network->no_of_layers > 1)
      send_layer(this_network, 0);

   if ( this_network->no_of_layers > 1)
 for (counter=1; counter< (this_network->no_of_layers-1); ++counter)
{

       process_layer(this_network->this_layer[counter]);
       send_layer(this_network, counter);
     }


  /* Finally the last layer and done ...and handle for one layer only.

*/
  if ( this_network->no_of_layers > 1)
process_layer(this_network->this_layer[this_network->no_of_layers
-1]);

}  /* end of PROCESS_NETWORK_B  */
```

```
     will be created (By me of course) so it could be set as a process
/*   Becker Co., Inc 1989 (C)                                          */
/*-------------------------------------------------------------------*/


#include <stdio.h>
#include  "main_def.h"

main(argc, argv)
int argc;
char **argv;

{ /* begin of MAIN */

   run_plain(argc,argv);
   exit(0);

}  /* end of MAIN */


MAIN DEF.H

/*
    The neuron will process   ----
                              \                    exp
                              /      w ( x   - x  )
                              ----     i    i    o


*/
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/mpadvise.h>

extern int errno;
extern char *sys_errlist[];
static int shmemtemp;
static int smneurtemp;

typedef struct
       neuron_type{
         int     no_of_inputs;
         double  *input_list;  /* can be impreoved..for time being
this
                                          is included...*/
         double  *weight_list;  /* the actual weights of neurons*/
         double  *delwgt; /* delta weight changes of neurons B.b.*/
```

```
          double  x_zero;
          double  exp;
          double  threshold;
          double  (* transfer_function)();
                /* any function can be placed here..cool */
          double *output;
          double *error; /*added BB */

          double *delta; /*added BB*/
          double *bias; /*added BB*/
          double *delbias; /*added BB*/
                  }NEURON;
typedef NEURON *NEURON_PTR;


typedef struct
     layer_type{
       int     no_of_neurons;  /* on this the layer  */
       NEURON_PTR *this_neuron;   /*pointer to neuron  pointers
                                              on layer */
          } LAYER;
typedef LAYER *LAYER_PTR;


typedef struct
     network_type{
          double  *net_input_list;    /* to have structure clear */
          double  *net_output_list;
          int     no_of_layers;   /* on this network  */
          LAYER_PTR  *this_layer; /* pointer to layer  pointers
                                              on network */
            } NETWORK;
typedef NETWORK *NETWORK_PTR;

NETWORK   current_network;   /* global holding current network */


typedef struct
          set_up{
              int    no_of_inputs;
              int    no_of_layers;
              int    *neurons_per_layer;
                } SET_UP_TYPE;


SET_UP_TYPE    network_set_up;


/*  Now define the global variables, they are not pointers because  */
/*  when the program gets bigger compiler goes crazy and cannot store
*/  string..*/
```

```
char    input_file_name[80], output_file_name[80],
        network_file_name[80], this_buffer[80];


int     global_cursor_y,  global_cursor_x;


BPTRN P.C
/*-------------------------------------------------------------------*/
/*   Module Name: Back_prop_train.c MODIFIED FOR PIPLEINE  bptrnp.c*/

/*   ============================                                   */
/*      Becker Co., Inc. 1989  (C)                                  */
/*-------------------------------------------------------------------*/

/* NOTE:  The field output of a neuron cell will be used to store the
error feedback..just temporarily for next interation..
*/

#include  <curses.h>
#include  "main_def.h"

#define forkflag misc[0]
#define parentflag misc[1]
#define doneflag misc[2]
#define procnetflag misc[3]
#define checkerrflag misc[4]
#define erroroutflag misc[5]

#define patterr lumped[0]
#define toterr lumped[1]

typedef struct  func_pass {
                            double   value;
                            double   thr;
                            double   (* this_function)();
                          }  FUNC_ARG_TYPE;

FILE *fopen(), *fp;

extern int *misc;
extern double *lumped;
int function, pid;

adjust_bp_error(this_network)
NETWORK  *this_network;

/* from the last layer will start backpropagating errors and adjusting

weights as it goes toward the first layer. Assumes that the error is
already backpropagated and stored on <.output> field of each neuron on
```

```
layer ...*/

{   /* begin of ADJUST_BP_ERROR  */
   int    i,  looper, neuron_loop;
       while(!checkerrflag);
       checkerrflag = 0;

  for ( i = this_network->no_of_layers - 1 ;
        i >= 0 ;   /*don't adjust first layer on loop */
        --i)
   {

  for ( neuron_loop = 0;
        neuron_loop < this_network->this_layer[i]->no_of_neurons;
        ++neuron_loop)

    {

     for (looper = 0;
          looper < this_network->this_layer[i]-
>this_neuron[neuron_loop]->no_of_inputs;
          ++looper)
     {

         this_network->this_layer[i]->this_neuron[neuron_loop]-
>delwgt[looper] =
            (0.5 * this_network->this_layer[i]-
>this_neuron[neuron_loop]->delta[0] *
            this_network->this_layer[i]->this_neuron[neuron_loop]-
>input_list[looper]) +
            (0.9 * this_network->this_layer[i]-
>this_neuron[neuron_loop]->delwgt[looper]);


      this_network->this_layer[i]->this_neuron[neuron_loop]-
>weight_list[looper] +=
       this_network->this_layer[i]->this_neuron[neuron_loop]-
>delwgt[looper];

         } /* end of looper for */
  } /* end of neuron_loop for */
 } /* end of i layer for loop */
}  /* end of ADJUST_BP_ERROR  LAYER  */


check_error_output(this_network)
NETWORK   *this_network;
/* Compare the output of net on training if it's on limit the stop
   and also update the output field on last layer of network.
   Returns 1 if output vector is on range to be considered close to
...*/
{   /* begin of  CHECK_ERROR_OUTPUT   */
```

```
   int  looper, neuron_loop;
   double temp_err, temp, sigmoid(), derivative();

   int inner, outer;
   int lay, i, j, inputs;
   FUNC_ARG_TYPE  dummy;

if(!forkflag){
  forkflag = 1;
  errno = 0;
  pid = fork();
  if(errno >0)
  fprintf(stderr, "Trouble with fork, errno = %3d, which means %s\n",
errno, sys_errlist[errno]);
  errno  = 0;
} /* end of not fork */


if(pid == 0){
errno = 0;
function = mpadvise(MPA_CPU_SET, 4);
if(errno >0)
fprintf(stderr, "Trouble with fork, errno = %3d, which means %s\n",
errno, sys_errlist[errno]);
errno = 0;
forkagain:

if(doneflag){
forkflag = 0;
kill(pid, SIGKILL);

exit(0);

} /* end if done */
   for ( outer =0;
         outer < this_network->no_of_layers; ++outer)
    for( inner=0; inner< this_network->this_layer[outer]-
>no_of_neurons; ++inner)
   this_network->this_layer[outer]->this_neuron[inner]->error[0]   =
0.0;

while (!procnetflag);
procnetflag = 0;

   patterr = 0.0;

for(looper = 0;
    looper < this_network->this_layer[this_network->no_of_layers - 1]
             ->no_of_neurons; ++ looper)
{

temp_err =  this_network->this_layer[this_network->no_of_layers - 1]-
```

```
>this_neuron[looper]
   ->error[0] = this_network->net_output_list[looper] -
         this_network->this_layer[this_network->no_of_layers - 1]
               ->this_neuron[looper]->output[0];

patterr += temp_err*temp_err;
}

toterr += patterr;

for(lay = this_network->no_of_layers -1; lay >=0; lay --)
   {
   for(i=0;i<this_network->this_layer[lay]->no_of_neurons; i++)
   {
 temp = this_network->this_layer[lay]->this_neuron[i]->bias[0];

   for(j=0;j<this_network->this_layer[lay]->this_neuron[i]-
>no_of_inputs; j++)
    {
     temp+= (this_network->this_layer[lay]->this_neuron[i]-
>input_list[j])

          *(this_network->this_layer[lay]->this_neuron[i]-
>weight_list[j]);
    }
   dummy.value = temp;
   dummy.thr =  this_network->this_layer[lay]->this_neuron[i]-
>threshold;
   dummy.this_function = sigmoid;
   temp = derivative(dummy);
temp = this_network->this_layer[lay]->this_neuron[i]->output[0];
   this_network->this_layer[lay]->this_neuron[i]->delta[0] =
   this_network->this_layer[lay]->this_neuron[i]->error[0] * temp *
(1.0 - temp);

if(lay < this_network->no_of_layers - 1)
checkerrflag = 1;

   for(j=0;j<this_network->this_layer[lay]->this_neuron[i]-
>no_of_inputs; j++)
   {
    if (lay ==0)
      inputs = 1;
    else
    {
    this_network->this_layer[lay-1]->this_neuron[j]->error[0] +=
    this_network->this_layer[lay]->this_neuron[i]->delta[0] *
    this_network->this_layer[lay]->this_neuron[i]->weight_list[j];

/*fprintf(fp, "error %f\n",this_network->this_layer[lay-1]-
>this_neuron[j]->error[0]);*/
} /* end else */
```

```
    } /* end j for loop */
} /* end i for loop */
} /* end lay for loop */
checkerrflag = 1;

  for ( i = this_network->no_of_layers - 1 ;
        i >= 0 ;  /* don't adjust first layer on loop  */
        --i)
   {

  for ( neuron_loop = 0;
        neuron_loop < this_network->this_layer[i]->no_of_neurons;
        ++neuron_loop)
    {
this_network->this_layer[i]->this_neuron[neuron_loop]->delbias[0] =
0.5 * this_network->this_layer[i]->this_neuron[neuron_loop]->delta[0]
+
0.9 * this_network->this_layer[i]->this_neuron[neuron_loop]-
>delbias[0];

this_network->this_layer[i]->this_neuron[neuron_loop]->bias[0] +=
this_network->this_layer[i]->this_neuron[neuron_loop]->delbias[0];

  } /* end of neuron_loop for */
} /* end of i layer for loop */

checkerrflag = 1;

while(!parentflag);
parentflag = 0;
goto forkagain;
} /* end of pid = zero */
if(pid != 0) return;

}   /* end of CHECK_ERROR_OUTPUT   */

back_prop_train(this_network, this_alpha)
NETWORK   *this_network;
double    this_alpha;
/* Assumes that data is already placed on the net_output/input_list
   vector on network... This function is for a plain terminal*/
{   /* begin of BACK_PROP_TRAIN   */

  extern   char *read_string(), *check_file();
  FILE     *temp, *fopen();
  int      loo, cursor_y, cursor_x, looper, dummy, no_of_patterns,
no_of_passes=0;
  int      fine, no_trials, t, done, i, step = 0, trials = 0;
  register double temp_err;

  int readit = 0;
```

```
  char     *fname, buff[80], this_choice;
  WINDOW   *back_screen, *error_screen , *create_window();

  parentflag = 0;
  forkflag = 0;
  doneflag = 0;

  initscr();
  back_screen = create_window(20,70, 4,4);
  error_screen = create_window(2,70, 22,4);


  insert_string( back_screen,
  "Will the patterns be entered from keyboard/file (k/f)?",1,2,0);
  wrefresh(back_screen);
  getyx( back_screen, cursor_y, cursor_x);
  this_choice = read_key_stroke(back_screen);

  if( this_choice != 'f' && this_choice != 'k')
  {
     error_message(back_screen, error_screen ,"Wrong input..",
                   cursor_y, cursor_x);
     this_choice = read_key_stroke(back_screen);
     werase(error_screen);
  }
  wrefresh(error_screen);
  insert_string(back_screen,"What is the pattern file name?",4,2,0);
  wrefresh(back_screen);

  getyx(back_screen,cursor_y, cursor_x);
  fname = read_string(back_screen);
  fname = check_file(fname, "r", back_screen, error_screen,
                                 cursor_y, cursor_x);
  strcpy(buff ,fname);

  check_pattern_file( buff, this_network);

  temp = fopen(buff, "r");

insert_string(back_screen, "For how many trials do you want to
train?",
6,2,0);
wrefresh(back_screen);
getyx(back_screen, cursor_y, cursor_x);
no_trials   =   get_integer(back_screen,   error_screen,   cursor_y,
cursor_x);
wrefresh(back_screen);
getyx(back_screen, cursor_y, cursor_x);
step = answer_yes_no(back_screen, error_screen,
    "Do you want to see all the steps (y/n)?", cursor_y+1, 1);
wrefresh(back_screen);
trials = answer_yes_no(back_screen, error_screen,
```

```
 "Do you want to see all the trials (y/n)?", cursor_y+2, 1);
werase(back_screen);
     print_network_stat(this_network, back_screen);
do_it_again:

for(t = 0; t < no_trials; ++t)
{
     no_of_passes += 1; /* incrememt the counter of the trainings */

if (trials){
 wrefresh(back_screen);

     wmove(back_screen, 5,4);

     wprintw(back_screen,
            "Network has gone through  %d  trials \n", no_of_passes);

}
if(!readit){

   rewind(temp);
   fscanf(temp, "%d\n", &no_of_patterns);
   fscanf(temp , "%d\n" , &dummy);  /* skip next 2 values on file..*/
   fscanf(temp , "%d\n" , &dummy);
}/*  end of if */
 toterr = 0.0;
   for( looper =0;
        looper < no_of_patterns;
        ++looper)
   {
if (trials){
      wmove(back_screen, 6,4);
      wprintw(back_screen,

     "Training file has  %d   patterns, current pattern is  %d\n",
                  no_of_patterns ,  looper + 1);
}
  if (!forkflag){

         read_network_io(this_network, temp);

         check_error_output(this_network);

  } /* end if*/
     process_network_b(this_network);
     procnetflag = 1;

if(trials){

       for(loo = 0;
           loo  <this_network->this_layer[this_network->no_of_layers
-1]
```

```
                    ->no_of_neurons;
            ++loo)
        {
          wmove( back_screen, 9 + loo , 2);
          wprintw(back_screen, " Item[ %d ] = %f",
           loo,   this_network->this_layer[this_network->no_of_layers
-1]
                                    ->this_neuron[loo]->output[0]);
    wmove( back_screen, 9 + loo , 25);
          wprintw(back_screen, "Desired Item[ %d ] = %f\n",
                 loo,   this_network->net_output_list[loo]);
         }
} /* end of if trials */
if(trials)
{
      wmove(back_screen, 7,4);
      wprintw(back_screen, "patterr = %f", patterr);
      wmove(back_screen, 7,24);
      wprintw(back_screen, "Toterr = %f\n", toterr);

} /* end of trial if */
if(looper != no_of_patterns - 1)
read_network_io(this_network, temp);
else {
read_network_io(this_network, temp);
    rewind(temp);
   fscanf(temp, "%d\n", &no_of_patterns);
   fscanf(temp , "%d\n" , &dummy);
   fscanf(temp , "%d\n" , &dummy);
read_network_io(this_network, temp);
readit = 1;
}

       adjust_bp_error(this_network);
       parentflag = 1;


if(step){

       wmove(back_screen, 16,5);
       wprintw(back_screen,  " %d BP iterations \n", no_of_passes);
       wprintw(back_screen,   "Press   Any   key   to   continue",
no_of_passes);
       wrefresh(back_screen);
       getch();
}
  } /* end of no of patterns FOR  */

     if (toterr < 0.04){
     fine = 1;
        break;
        }
```

```
} /* end of no_trials FOR */
parentflag = 1;
kill(pid, SIGKILL);
if(!trials){
 wrefresh(back_screen);

     wmove(back_screen. 5,4);

     wprintw(back_screen,
            "Network has gone through  %d  trials \n", no_of_passes);

      wmove(back_screen, 6,4);
      wprintw(back_screen,
      "Training file has  %d   patterns, current pattern is  %d\n",
                  no_of_patterns ,  looper);
       for(loo = 0;
           loo  <this_network->this_layer[this_network->no_of_layers
-1]
                    ->no_of_neurons;
           ++loo)
       {
         wmove( back_screen, 9 + loo , 2);
         wprintw(back_screen, " Item[ %d ] = %f",
          loo,   this_network->this_layer[this_network->no_of_layers
-1]
                                    ->this_neuron[loo]->output[0]);
    wmove( back_screen, 9 + loo , 25);
         wprintw(back_screen, "Desired Item[ %d ] = %f\n",
                loo,   this_network->net_output_list[loo]);

        }

    wmove(back_screen, 7,4);
      wprintw(back_screen, "patterr = %f", patterr);
      wmove(back_screen, 7,24);
      wprintw(back_screen, "Toterr = %f\n", toterr);
} /* end of not trials IF */

if(fine == 1){
wmove(back_screen, 16,0);
 wprintw(back_screen, "FINISHED, solution found after %d training
sequences\n", no_of_passes);

 wrefresh(back_screen);
}


    wmove(back_screen, 18,5);
    getyx(back_screen,cursor_y, cursor_x);

if(answer_yes_no(back_screen, error_screen,
```

```
                "Do you want stats (y/n)?", cursor_y, cursor_x) ==
1){
      system ("clear");
      rstats(stdout);
      wmove(back_screen, 17,0);
      getyx(back_screen, cursor_y, cursor_x);
}
   if(  answer_yes_no(back_screen, error_screen,
 " Would you like to train again (y/n)?", cursor_y, cursor_x) == 1){

        system("clear");
     initscr();
        if (done==1)
           no_of_passes = 0;
                readit = 0;
                forkflag = 0;
     goto do_it_again;
   }
 endwin();
}   /* end of BACK_PROP_TRAIN  */
```

NEURTULP.C

```
/*-------------------------------------------------------------*/
/*Module Name:  neuron_tool.c MODIFIED FOR PIPELINE neurtulp.c  */
/*=========================                                    */
/*Set of routines to handle the actual network operations      */
/* it's device independat,i.e. just carries out the math and   */
/*connections, can be called from any screen/window environment */
/*   Becker Co., Inc. 1989 (C)                                 */
/*-------------------------------------------------------------*/


#include <stdio.h>
#include <malloc.h>
#include "main_def.h"

#define TF_VALUE_L  0.0
#define TF_VALUE_H  1.0



int lumpid, lumppage, pid;
unsigned lumpsize;
double *lumped = NULL;


int miscid, miscpage;
unsigned miscsize;
int *misc = NULL;
```

```
typedef struct  func_pass {
                            double   value;
                            double   thr;
                            double   (* this_function)();
                          }  FUNC_ARG_TYPE;

FILE *fopen(), *fp;

/*
  This set of routines create (dynamically) a neural
  network of any size the only limitation is the machine
  that is running it.

  It will be developed as device independent as possible so that
  to make it run on any machine all it would be needed will be
  the drivers for that particular machine (for the graphics ouput)

     by Becker Co, Inc (1989) (C)
*/

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 1
-----------------------------------*/

NEURON create_neuron( no_of_input)
int no_of_input;
{
   NEURON  temp_neuron;
   int     looper;
   double  sharp(), sigmoid();
/*   temp_neuron.input_list = (double *)calloc(no_of_input,
                                                sizeof(double));
   temp_neuron.weight_list = (double *)calloc(no_of_input,
                                                sizeof(double));
   temp_neuron.delwgt = (double *)calloc(no_of_input,
                                                sizeof(double));

*/

temp_neuron.input_list = &lumped[shmemtemp];
shmemtemp += no_of_input;
temp_neuron.weight_list = &lumped[shmemtemp];
shmemtemp += no_of_input;
temp_neuron.delwgt = &lumped[shmemtemp];
shmemtemp += no_of_input;

temp_neuron.output = &lumped[shmemtemp];
shmemtemp += 1;
temp_neuron.error = &lumped[shmemtemp];
shmemtemp += 1;
temp_neuron.delta = &lumped[shmemtemp];
shmemtemp += 1;
```

```
temp_neuron.bias = &lumped[shmemtemp];
shmemtemp += 1;
temp_neuron.delbias = &lumped[shmemtemp];
shmemtemp += 1;


   /* Now that we allocate space for the weights..initialize randomly
*/
   for ( looper =0 ; looper < no_of_input ; ++looper)
{     temp_neuron.weight_list[looper] =( rand()* 6.1035E-5) - 1.0;
     temp_neuron.delwgt[looper] = 0.0;
}

   /* compiler doen't like undefined values..so intialize everything
*/

   temp_neuron.no_of_inputs = no_of_input;
   temp_neuron.x_zero = 0.0;
   temp_neuron.exp = 1.0;
   temp_neuron.threshold = 0.0;
   temp_neuron.transfer_function = sigmoid ; /* default sharp TF */

   temp_neuron.output[0]    = 0.0;
   temp_neuron.error[0] = 0.0;
   temp_neuron.delta[0] = 0.0;
   temp_neuron.bias[0] = 0.0;
   temp_neuron.delbias[0] = 0.0;


   return(temp_neuron);
}
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 2
----------------------------------*/

LAYER create_layer( layer_no, no_of_neurons, prev_layer_no_of_neurons)
int layer_no;
int no_of_neurons;
int prev_layer_no_of_neurons;

{  /* begin of CREATE_LAYER */
   NEURON      create_neuron();

   LAYER       temp_layer;
   int         i;
   NEURON      *dummy_neuron;  /* memory space MUST be allocated to
hold
                                the actual neuron space  */

    /*  Create dynamically the pointers to neurons*/
    temp_layer.this_neuron = (NEURON_PTR *)calloc(no_of_neurons,

sizeof(NEURON_PTR));
```

```
  dummy_neuron  = (NEURON *)calloc(no_of_neurons, sizeof(NEURON) );


    temp_layer.no_of_neurons = no_of_neurons;


    /* Now create the actual neurons  */
    for (i=0; i < no_of_neurons ; ++i)
    {
       dummy_neuron[i] = create_neuron(prev_layer_no_of_neurons);
       temp_layer.this_neuron[i] = &dummy_neuron[i];
    }

    return(temp_layer);


}  /* end of CREATE_LAYER */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 3
-----------------------------------*/

NETWORK create_network(net_set_up)
SET_UP_TYPE net_set_up;
{  7* begin of CREATE_NETWORK */

     NETWORK network_temp;
     int     counter;
         LAYER *dummy_layer;

int i, totneuron, connects;
totneuron = 0;
connects = 0;

/*the following calcs the size of the network then creates enough
shared memory space to handle the parallel procesing */

for(i = 0; i < net_set_up.no_of_layers; i++)
{
     totneuron += net_set_up.neurons_per_layer[i];
     if (i == 0)
        connects = net_set_up.no_of_inputs *
net_set_up.neurons_per_layer[i];
     else
        connects += net_set_up.neurons_per_layer[i-1] *
                    net_set_up.neurons_per_layer[i];
} /* end of for */

connects= (net_set_up.neurons_per_layer[0] *2 ) +
(net_set_up.neurons_per_layer[(net_set_up.no_of_layers - 1)]*2) +
(connects * 3) + (totneuron * 5) + 4;
```

```
errno = 0;

lumpsize = (unsigned)(sizeof(double) *  connects );
lumpid = shmget(0, lumpsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "lumped errno is %3d, which means %s\n", errno,
sys_errlist[errno]);

errno = 0;
lumppage = calc_page(lumpsize);
lumped = (double *) shmat(lumpid, lumppage, 0);
if (errno > 0)
fprintf(stderr, "In lumped  %s\n", sys_errlist[errno]);
for( i = 0; i< connects ; i++)
     lumped[i] = 0.0;

/*create enough space for ten flags to be used in bp_train.c-- see
define section */
miscsize = (unsigned)(sizeof(int) *  10 );
miscid = shmget(0, miscsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "misc errno is %3d, which means %s\n", errno,
sys_errlist[errno]);
errno = 0;
miscpage = calc_page(miscsize);
misc = (int *) shmat(miscid, miscpage, 0);
if (errno > 0)
fprintf(stderr, "In misc  %s\n", sys_errlist[errno]);
for( i = 0; i< 10 ; i++)
     misc[i] = 0;

shmemtemp = 2;
smneurtemp = 0;

     /* Create fist the pointers to the layers of the network */
     network_temp.this_layer = (LAYER_PTR
*)calloc(net_set_up.no_of_layers,sizeof(LAYER_PTR));
    dummy_layer = (LAYER *)calloc(net_set_up.no_of_layers,
sizeof(LAYER) );


     network_temp.no_of_layers = net_set_up.no_of_layers;

     for ( counter=0; counter < net_set_up.no_of_layers ; ++counter)
        {
           if ( counter == 0)   /* first layer */
        {
              dummy_layer[counter] = create_layer(counter,
                                net_set_up.neurons_per_layer[counter],
                                 net_set_up.no_of_inputs);
                   network_temp.this_layer[counter]    =
&dummy_layer[counter];
```

```
        }
           else
          {
               dummy_layer[counter] = create_layer( counter,
                            net_set_up.neurons_per_layer[counter],
                            net_set_up.neurons_per_layer[counter-1]);
               network_temp.this_layer[counter] =
&dummy_layer[counter];
             }
        }


        /* Now create the input and output array (dynamically) as
requested
           by user...to be used to train or output nicely*/

      network_temp.net_input_list = &lumped[shmemtemp];
      shmemtemp+= net_set_up.neurons_per_layer[0] + 10;

      network_temp.net_output_list = &lumped[shmemtemp];
 shmemtemp+=   net_set_up.neurons_per_layer[(net_set_up.no_of_layers
-1)]
+ 10;

        return ( network_temp);

}  /* begin of CREATE_NETWORK */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 4
-----------------------------------*/
/*     Now a set of routines to read  and write the current
*/
/*     network ..so work is not lost
*/
/*-----------------------------------------------------------------
-*/
save_neuron(this_neuron, this_file)
NEURON   *this_neuron;
FILE     *this_file;

/* assumes that file was already open in calling function..just saves
   the neuron in order..i.e all fileds  */
{   /* begin of SAVE_NEURON   */
   int temp_index;

   fprintf(this_file,"%d\n", this_neuron->no_of_inputs);
   fprintf(this_file,"%f\n", this_neuron->bias[0]);

/* finally save the weights  */

   for ( temp_index =0; temp_index < this_neuron->no_of_inputs;
         ++temp_index)
```

```
   {
              f p r i n t f ( t h i s _ f i l e , " % f \ n " ,
this_neuron->weight_list[temp_index]);

   }

}   /* end of SAVE_NEURON     */
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 5
----------------------------------*/

save_network(this_network, file_name)
NETWORK   *this_network;
char      *file_name;

{  /* begin of SAVE_NETWORK  */
   int   layer_index, neuron_index;
   FILE  *this_file;

   if ( (this_file = fopen(file_name , "w") ) == NULL)
   {

     printf("Sorry cannot write on this file...\n");
     exit(1);
   }

  /* start by saving the network set up..*/
   fprintf(this_file, "%d\n" , this_network->this_layer[0]-
>this_neuron[0]
                                ->no_of_inputs);
   fprintf(this_file, "%d\n" , this_network->no_of_layers);

   for ( layer_index =0;
         layer_index < this_network->no_of_layers;
         ++layer_index)
   {
 fprintf( this_file, "%d\n"/* = neurons on layer-------- [ %d ] \n"*/,
             this_network->this_layer[layer_index]->no_of_neurons/*,
              layer_index*/);
   }

   /* Now save the network ..every neuron from 0 -> no  */
   for ( layer_index =0;
         layer_index < this_network->no_of_layers;
         ++layer_index)
  {
      for ( neuron_index =0;
      neuron_index < this_network->this_layer[layer_index]-
>no_of_neurons;+neuron_index)
      {
          save_neuron(this_network->this_layer[layer_index]->
                      this_neuron[neuron_index],  this_file );
      }
```

```
  }


}  /* end of SAVE_NETWORK  */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 6
-----------------------------------*/


read_neuron(this_neuron, this_file)
NEURON   *this_neuron;
FILE     *this_file;

/* assumes that file was already open in calling function..just saves
   the neuron in order..i.e all fields  */
{   /* begin of READ_NEURON   */
   int temp_index;

   fscanf(this_file,"%d\n", &this_neuron->no_of_inputs);
   fscanf(this_file,"%lf\n", &this_neuron->bias[0]);

   /* finally read the weights  */


   for ( temp_index =0; temp_index < this_neuron->no_of_inputs;
         ++temp_index)
   {
       fscanf(this_file,"%lf\n", &this_neuron-
>weight_list[temp_index]);
   }

}   /* end of READ_NEURON     */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 7
-----------------------------------*/

read_network(this_network, file_name)
NETWORK   *this_network;
char  *file_name;

/* Note that to be on the safe side when calling this function it's
   better... BELIEVE ME..to send in a buffer[80] instead of just the
   pointer for 'file_name' otherwise it's not reliable  */

{  /* begin of READ_NETWORK  */
   int          layer_index, neuron_index;
   FILE         *fopen(), *this_file;
   SET_UP_TYPE  temp_set_up;
   NETWORK      create_network();

  if ( (this_file = fopen(file_name , "r")) == NULL)
```

```
  {
     printf("Sorry cannot read on this file...\n");
     exit(1);
   }

   rewind(this_file);

  /* start by reading the network set up..*/
   fscanf(this_file, "%d\n", &temp_set_up.no_of_inputs);
   fscanf(this_file, "%d\n", &temp_set_up.no_of_layers);
   temp_set_up.neurons_per_layer = (int *)calloc(
temp_set_up.no_of_layers,
        sizeof(int)  );
   for ( layer_index =0;
         layer_index < temp_set_up.no_of_layers;
         ++layer_index)
   {
      fscanf( this_file, "%d\n ",
              &temp_set_up.neurons_per_layer[layer_index]);
   }


   /* Once the set up is there..create the memory space to hold the
      network to be read...*/
      *this_network = create_network(temp_set_up);

   /* Now read the network ..every neuron from 0 -> no  */
   for ( layer_index =0;

         layer_index < this_network->no_of_layers;
         ++layer_index)
  {
      for ( neuron_index =0;
  neuron_index<this_network->this_layer[layer_index]->no_of_neurons;
            ++neuron_index)
      {
          read_neuron(this_network->this_layer[layer_index]->
                                        this_neuron[neuron_index],
                      this_file );
      }

  }


}  /* end of READ_NETWORK  */


check_pattern_file(file_name, for_this_network)
char     *file_name;
NETWORK  *for_this_network;
/*
   Will scan the file with the pattern(s) and check that all data is
```

```
   consistent..so when training starts it won't hang ...Assumes that
the
   file exits when called..

    Outputs 0 on success, 1 on failure..
*/
{   /* begin of CHECK_PATTERN_FILE */

   FILE  *this_file, *fopen();
   int      no_of_patterns, no_of_inputs, no_of_outputs,
            looper, counter ;
double   temp_value;

   /* Check first if the file exists */
   if ( (this_file = fopen(file_name , "r")) == NULL)
   {
     printf("Sorry cannot read this file...\n");
     exit(1);
   }

   rewind(this_file);

   /* Now that the file is there check if patterns are correct,
       i.e. dimesions and format  */
   fscanf(this_file, "%d\n", &no_of_patterns);
   fscanf(this_file, "%d\n", &no_of_inputs);
   fscanf(this_file, "%d\n", &no_of_outputs);
   if ( (no_of_inputs != for_this_network->this_layer[0]-
>this_neuron[0] ->no_of_inputs )   ||
         (no_of_outputs != for_this_network-
>this_layer[for_this_network ->no_of_layers -1]->no_of_neurons) )

   {
     printf("Sorry patterns dimensions do not match network's...\n");
     exit(1);
   }

   /* Finally check that file contains patterns correctly
      doesn't check when incomplete data... */

   for (looper =0; looper < no_of_patterns; ++looper)
   {
        for ( counter =0;
              counter < no_of_inputs;
              ++ counter)
          if ( fscanf(this_file,"%lf\n", &temp_value) == EOF)
           {
              printf( " Sorry..incomplete data  on file...\n");
              exit(1);
           }

        for (counter=0; counter < no_of_outputs ; ++counter)
```

```
        {
           if ( fscanf(this_file,"%lf\n", &temp_value) == EOF)
           {
              printf( " Sorry..incomplete data  on file...\n");
              exit(1);
           }
          if (temp_value != TF_VALUE_H && temp_value != TF_VALUE_L)
          {
            printf(" Outputs have different dimension \n");
            exit(1);
          }
        }
   }
   /* If everything went correct then the file has right information
      return succesful code..*/
   return(0);

}   /* end of CHECK_PATTERN_FILE */


read_network_io(this_network,  this_file)
NETWORK  *this_network;
FILE     *this_file;
/* Assumes that file already exists and that it's already open so
   it's just ready to read the input and ouput from current position
    on file...
*/
{  /*  begin of READ_NETWORK_IO  */
   int looper;

   /* read the input vector and place  it on network */
   for (looper=0;
        looper < this_network->this_layer[0]->this_neuron[0]
                                    ->no_of_inputs  ;

        ++looper)
      fscanf(this_file, "%lf\n", &this_network-
>net_input_list[looper]);

   /* read the output vector and place  it on network */
   for (looper=0;
  looper < this_network->this_layer[this_network->no_of_layers -1]
                                      ->no_of_neurons ;
        ++looper)
      fscanf(this_file, "%lf\n", &this_network-
>net_output_list[looper]);

}  /*  end of READ_NETWORK_IO  */


/*
    Now a set of transfer functions that can be used as transfer
```

```
function
*/

double sharp(this_x, this_threshold)
double   this_x;
double   this_threshold;

{   /* begin of SHARP */
  if (  this_x >= this_threshold)
    return(TF_VALUE_H);
  else
   return(TF_VALUE_L);
}  /* end of SHARP   */


double sigmoid( this_x, this_threshold)
double   this_x;
double   this_threshold;
{   /* begin of SIGMOID  */
   double exp();
  double limit = 12.0;

  /* just in case convert to double everything..*/
   if ( this_x - this_threshold > limit)
      return( 0.9999998);
   else
     if ( this_x - this_threshold < -limit)
      return( 0.0000012);


    else
   return(  ( 1.0/(1.0 +  exp((-1.0)* (this_x - this_threshold)))
)       );
}   /* end of SIGMOID   */


double  derivative(this_func)

FUNC_ARG_TYPE   this_func;
{  /* begin of DERIVATIVE  */
   double  temp, ddelta = 1.0E-10;

   temp =  ( this_func.this_function(this_func.value + ddelta,
this_func.thr) -
             this_func.this_function(this_func.value, this_func.thr)
           ) / ddelta;

   return(temp);

}  /* end of DERIVATIVE   */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 7
```

```
-----------------------------------*/
/*                                                                  */
/*       Now a set of routine to process the network                */
/*                                                                  */

send_neuron_output(this_value, on_this_network,  from_layer_index,
                   from_neuron_index, to_neuron_index)
double  this_value;
NETWORK *on_this_network;
int     from_layer_index;
int     from_neuron_index;
int     to_neuron_index;

/* Calling routine should check that the last layer doesn't send
output
  or this routine will do nothing...*/

{  /* begin of SEND_NEURON_OUTPUT   */

   if ( (from_layer_index +1) >= on_this_network->no_of_layers)
     {
        printf("  Sorry no next layer to send values to..");
        exit(1);
     };

   on_this_network->this_layer[from_layer_index + 1]
              ->this_neuron[to_neuron_index]-
>input_list[from_neuron_index]=
              this_value;
}  /* end of SEND_NEURON_OUTPUT  */

/*--------------------------F .8 ----------------------------------*/

send_layer(on_this_network, this_layer_index)
NETWORK  *on_this_network;
int      this_layer_index;

 /* This function will send values from layer to next ..in (hidden)
layers,
    for last layer use other routine   */

{   /* begin of SEND_LAYER  */

    int  from_ctr, to_ctr, from_tmp, to_tmp;

   /* if ( (this_layer_index == 0) ||   */
      if   (this_layer_index >= (on_this_network->no_of_layers -1) )
    {
       printf(" Sorry cannot process this layer...");
       exit(1);
    }
```

```
  /* We waste a little memory (~4bytes) but speed is gained ...*/
  from_tmp = on_this_network->this_layer[this_layer_index]-
>no_of_neurons ;
  to_tmp   = on_this_network->this_layer[this_layer_index +1]-
>no_of_neurons ;

  for ( from_ctr =0; from_ctr < from_tmp; ++from_ctr)
    for ( to_ctr = 0; to_ctr < to_tmp; ++to_ctr)
   {
     send_neuron_output(
         on_this_network->this layer[this_layer_index]-
>this_neuron[from_ctr]->
                output[0],
         on_this_network,
         this_layer_index,
         from_ctr, to_ctr);

   }

;  /* end of SEND_LAYER  */
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 9
-----------------------------------*/

send_outer_layer(on_this_network, first_or_last)
NETWORK  *on_this_network;
int      first_or_last;

{   /* begin  of SEND_OUTER_LAYER  */
   int   counter, counter1, temp;

   switch( first_or_last)
  {
  case 0: temp = 0;
         for ( counter=0;
               counter<on_this_network->this_layer[0]->no_of_neurons

;
               ++counter)

           for ( counter1=0;
                 counter1 < on_this_network->this_layer[0]->
                 this_neuron[0]->no_of_inputs ; ++counter1)

          {
           on_this_network->this_layer[0]->this_neuron[counter]
               ->input_list[counter1]=
                     on_this_network->net_input_list[counter1];

          };

          break;
```

Appendix 5

Modified PDP Source Code for Hybrid Epoch-Pattern Method

GENHYB.C

```
/* This file is part of the PDP software package.  Copyright 1987 by
James L. McClelland and David E. Rumelhart. Please refer to licensing
information in the  file license.txt, which is in the same directory
with this source file and is included here by reference.*/

/* general.c
     Some general functions for PPD-pc package.
     First version implemented by Elliot Jaffe.
     Date of last revision:  8-12-87/JLM.
     Masscomp revisions:  30 OCT 90/ bob Bennington */

#include "general.h"
#include "command.h"
#include "variable.h"
#include <signal.h>
#ifdef MSDOS
#include <memory.h>   /* for memcpy() in erealloc in genhyb.c */
#include <process.h>  /* for system() in do_exec in command.c */
#endif
FILE * in_stream = stdin;
int     Interrupt_flag = 0;
int  single_flag = 0;
int  step_size;
int  random_seed;
char step_string[STRINGLENGTH];
struct Command_table   *Command;
extern int  dump_template ();
extern int  clear_display ();
extern int  update_display ();
extern int  redisplay ();
extern int  do_io ();
extern int  do_network ();
extern int  do_system ();
extern int  do_command ();
extern int  do_comfile ();
extern int  do_exec ();
extern int  set_log ();
extern int  run_fork();
extern float *wds;
extern int wdsid;
extern int lumpid;
extern float *lumped;
extern int *misc;
extern int miscid;
```

```
extern float *dwts;
extern int dwtsid;
extern float *wtss;

extern int wtssid;
extern int fnwtid;
extern int *fnwt;
extern float *wds;
extern float **wtwed;
extern int wtwedid;

int_handler() {
    int     int_handler ();

    (void) signal(SIGINT, int_handler);
    Interrupt_flag = 1;
}

#ifdef MSDOS
char *index(somestring,somechar)
char *somestring;
char somechar;
{
    return strchr(somestring,somechar);
}
#endif

char   *emalloc (n)          /* check return from malloc */
unsigned    n;
{
    char   *p,
           *malloc ();

    p = malloc(n);
    if (p == 0)
     put_error("out of memory");
    return p;
}

char   *erealloc (ptr,oldsize,newsize)        /* check return from
realloc*/
char *ptr;
unsigned    oldsize;
unsigned    newsize;
{
#ifndef MSDOS
    char   *realloc ();
    char   *p;

    p = realloc(ptr,newsize);
    if (p == 0)
     put_error("out of memory");
```

```
    return p;

#else (if MSDOS)

    char *malloc();
    char *p;


    p = malloc(newsize);
    if (p == 0)
        put_error("out of memory");
    if (ptr && p) {
     memcpy(p, ptr, oldsize);
     free(ptr);
    }
    return p;

#endif MSDOS
}

calc_page(size)   /* added 7 apr 89 RWB */
int size;         /* used to calc amount of space needed for */
{                 /* shared memory in increments of 4K pages */
int inc, pg, endof, xtra, nuend;
inc = 1;
pg = 4096;
endof = sbrk(0);
while (pg<= endof){
     pg = 4096 * inc;
     ++inc;
}
xtra = (pg - endof) + size;
nuend = sbrk(xtra);
return(pg);
}

startsame(s1, s2)            /* does s1 start the same as s2? */
char   *s1,
       *s2; {
    while (*s1 && *s2) {
     if (*s1++ != *s2++)
         return(0);
    }
    if(*s1 && !*s2)  /* if s1 is longer than s2 it should fail */
       return(0);
    return(1);
}

char   *strsave (s)
char   *s;
{
    char   *p,
```

```
            *emalloc ();

    if ((p = emalloc((unsigned)(strlen(s) + 1))) != NULL)
     (void) strcpy(p, s);
    return(p);
}


randint(low, high)

 int  low,high; {
    int     answer;
    float   randf;
    int     range;

    randf = rnd();
    range = high - low + 1;
    answer = randf * range + low;
    return(answer);
}

quit() {
   int r;
   char * str;
   str = get_command("Quit program? -- type y to confirm:  ");

   if (str && str[0] == 'y') {
         end_display();
           /* added 7 apr 89 detaches shared */
             /* memory  and deallocates its storage- RWB */
         shmdt(misc);
            shmdt(lumped);
         shmdt(fnwt);
            shmdt(wtss);
         shmdt(dwts);
         shmdt(wds);
         shmdt(wtwed);
            shmctl(fnwtid, IPC_RMID,0);
            shmctl(miscid, IPC_RMID,0);
            shmctl(lumpid, IPC_RMID,0);
            shmctl(wtwedid, IPC_RMID,0);
            shmctl(dwtsid, IPC_RMID,0);
            shmctl(wtssid, IPC_RMID,0);
            shmctl(wdsid, IPC_RMID,0);

     exit(0);
   }
   else
      return(CONTINUE);
}
stats() {    /* this function clears the screen then prints out */
clear_display(); /* the statistics package. It also returns the curser
```

```
*/
io_move(5,0);  /* back where it belongs so the command line will be */
io_refresh(); /* in the proper position  --31Jan 89 bob bennington */
     rstats(stdout);
     io_move (0,0);
     io_refresh();
     return(CONTINUE);

}

set_step() {
    char old_step_string[STRINGLENGTH];

    struct Variable *vp, *lookup_var();

    strcpy(old_step_string,step_string);

    vp = lookup_var("stepsize");
    change_variable("stepsize",vp);

    if (startsame(step_string,"nepochs"))
strcpy(step_string,"nepochs");
    else if (startsame(step_string,"epoch"))
strcpy(step_string,"epoch");
    else if (startsame(step_string,"pattern"))
strcpy(step_string,"pattern");
    else if (startsame(step_string,"ncycles"))
strcpy(step_string,"ncycles");
    else if (startsame(step_string,"cycle"))
strcpy(step_string,"cycle");
    else if (startsame(step_string,"update"))
strcpy(step_string,"update");
    else if (startsame(step_string,"default"))
                        strcpy(step_string,Default_step_string);
    else {
     strcpy(step_string,old_step_string);
     return(put_error("urecognized stepsize -- size not changed."));
    }
    set_stepsize();
    return(CONTINUE);
}

set_stepsize() {
    if (strcmp(step_string,"update") == 0) step_size = UPDATE;
    else if (strcmp(step_string,"cycle") == 0) step_size = CYCLE;
    else if (strcmp(step_string,"ncycles") == 0) step_size = NCYCLES;
    else if (strcmp(step_string,"pattern") == 0) step_size = PATTERN;
    else if (strcmp(step_string,"epoch") == 0) step_size = EPOCH;
    else if (strcmp(step_string,"nepochs") == 0) step_size = NEPOCHS;
}
```

```
init_general() {
    extern int     int_handler ();

    Interrupt_flag = 0;
    strcpy(step_string,Default_step_string);
    set_stepsize();
    init_commands();
    (void) signal(SIGINT, int_handler);
    (void) install_command("?", do_help, 0, 0);
    (void) install_command("disp/", do_command, BASEMENU, (int *)
DISPLAYMENU);
    (void) install_command("opt/",  do_command, DISPLAYMENU, (int *)
DISPLAYOPTIONS);
    (void) install_command("exam/", do_command, BASEMENU, (int *)
SETMENU);

    (void) install_command("get/",  do_command, BASEMENU, (int *)
GETMENU);
    (void) install_command("save/", do_command, BASEMENU, (int *)
SAVEMENU);
    (void) install_command("set/",  do_command, BASEMENU, (int *)
SETMENU);
    (void) install_command("config/", do_command, SETMENU, (int *)
SETCONFMENU);
    (void) install_command("env/",  do_command, SETMENU, (int *)
SETENVMENU);
    (void) install_command("mode/", do_command, SETMENU, (int *)
SETMODEMENU);
    (void) install_command("param/",do_command, SETMENU, (int *)
SETPARAMMENU);
    (void) install_command("state/", do_command, SETMENU, (int *)
SETSVMENU);
    (void) install_command("clear", clear_display, BASEMENU, 0);
    (void) install_command("do",  do_comfile, BASEMENU, 0);
    (void) install_command("log",   set_log, BASEMENU, 0);
    (void) install_command("quit",  quit, BASEMENU, 0);
    (void) install_command("run", do_exec, BASEMENU, 0);
    (void) install_command("stats", stats, BASEMENU, 0); /* added 30
Jan89 by bob bennington */
/*  (void) install_command("srand", random_seed, BASEMENU, 0); */
    (void) install_command("state", redisplay, DISPLAYMENU, 0);
    (void) install_var("seed", Int, (int *) & random_seed, 0,
0,SETPCMENU);
    (void) install_var("single", Int, (int *) & single_flag, 0,
0,SETPCMENU);
    (void) install_var("stepsize", String, (int *) step_string,0,
0,NOMENU);
    (void)   install_command("stepsize",set_step,SETPCMENU,(int   *)
NULL);
}

#ifdef MSDOS
```

```
sleep(n_sec)
int n_sec;
{
    int i,j;
    for (i = 0; i < (n_sec); i++)
        for (j = 0; j < 20000; j++);
}
#endif  MSDOS
```

BPH.C

```
/* file: bp.c  MODIFIED TO bph.c

     Do the actual work for the bp program.

     First version implemented by Elliot Jaffe

     Date of last revision:  8-12-87/JLM

HYBRID VERSION   Masscomp revisions
*/

#include "general.h"
#include "bp.h"
#include "variable.h"
#include "wgtsh.h"
#include "patterns.h"
#include "command.h"

#define forkflag misc[0]
#define compout1flag misc[1]
#define comperrorflag1 misc[2]
#define comperrorflag2 misc[3]
#define compwedflag1 misc[4]
#define parentflag misc[5]
#define doneflag misc[6]
#define patno misc[7]
#define compout2flag misc[8]
#define compwedflag2 misc[9]
#define pss lumped[6*nunits]
#define tss lumped[6*nunits +1]
#define momentum lumped[6*nunits +2]
/*#define lrate lumped[6*nunits +3]*/

char   *Prompt = "bp: ";
char   *Default_step_string = "epoch";
char grain_string[20] = "pattern";
boolean System_Defined = FALSE;
boolean lflag = 1;
boolean cascade = 0;
int     epochno = 0;
```

```
int     cycleno = 0;
int     nepochs = 500;
int  ncycles = 50;
/*int     patno = 0;*/
/*float   tss = 0.0;
float   pss = 0.0;*/
float   ecrit = 0.0;
float      crate = .05;
float      drate = .95;
float      gcor = 0.0;
int  follow = 0;
float  *netinput = NULL;
float  *activation = NULL;
float  *error = NULL;
float  *target = NULL;
float  *delta = NULL;
float  **dweight = NULL;
float  **pwed = NULL;
float  *dbias = NULL;
float  *pbed = NULL;
float   tmax = 1.0;
/*float   momentum = 0.9;*/

float      mu = .5;
int  tallflag = 0;

int    *misc = NULL;
extern float *lumped;
extern float **wtwed;
float *dwts = NULL;
float *wtss = NULL;
float *wds = NULL;

extern int read_weights();
extern int write_weights();

int  function, status, pid;
int miscid, miscpage;
unsigned miscsize;
int wtssid, wtsspage;
unsigned wtsssize;
int wdsid, wdspage;
unsigned wdssize;
int dwtsid,dwtspage;
unsigned dwtssize;
FILE *fopen(), *fp;

init_system() {
    int     strain (), ptrain (), tall (), test_pattern (),
reset_weights();
    int        get_unames(), set_lgrain(), cycle(), newstart();
    int change_lrate(), change_crate(), set_follow_mode();
```

```
    epsilon_menu = SETCONFMENU;

    init_weights();

    (void) install_command("strain", strain, BASEMENU,(int *) NULL);
    (void) install_command("ptrain", ptrain, BASEMENU,(int *) NULL);
    (void) install_command("tall", tall, BASEMENU,(int *) NULL);
    (void) install_command("test", test_pattern, BASEMENU,(int *)
NULL);
    (void) install_command("cycle", cycle, BASEMENU,(int *) NULL);
    (void)    install_command("reset",reset_weights,BASEMENU,(int
*)NULL);
    (void) install_command("newstart",newstart,BASEMENU,(int *)NULL);
    (void)  install_command("unames",  get_unames,  GETMENU,(int  *)
NULL);
    (void) install_command("patterns", get_pattern_pairs,
                                GETMENU,(int *) NULL);
    (void) install_var("lflag", Int,(int *) & lflag, 0, 0, SETPCMENU);
    (void) install_var("lgrain", String, (int *) grain_string,0,
0,NOMENU);
    (void) install_command("lgrain",set_lgrain,SETMODEMENU,(int *)
NULL);
    (void) install_var("follow", Int, (int *) & follow,0, 0,NOMENU);
    (void) install_command("follow",set_follow_mode,SETMODEMENU,(int
*)
NULL);

    (void) install_var("cascade", Int,(int *) & cascade, 0, 0,
SETMODEMENU);
    (void) install_var("nepochs", Int,(int *) & nepochs, 0, 0,
SETPCMENU);
    (void) install_var("ncycles", Int,(int *) & ncycles, 0, 0,
SETPCMENU);
    (void) install_var("epochno", Int,(int *) & epochno, 0, 0,
SETSVMENU);
/*    (void) install_var("patno", Int,(int *) & patno, 0, 0,
SETSVMENU); */
    (void) install_var("cycleno", Int,(int *) & cycleno, 0, 0,
SETSVMENU);
    init_pattern_pairs();
/*   (void) install_var("pss", Float,(int *) & pss, 0, 0, SETSVMENU);
    (void) install_var("tss", Float,(int *) & tss, 0, 0, SETSVMENU);*/
    (void) install_var("gcor", Float,(int *) & gcor, 0, 0, SETSVMENU);
/*    (void) install_var("momentum", Float,(int *)
&momentum,0,0,SETPARAMMENU);*/
    (void) install_var("mu", Float,(int *) &mu,0,0,SETPARAMMENU);
    (void) install_command("lrate", change_lrate, SETPARAMMENU, (int
*)
NULL);
    (void) install_command("crate", change_crate, SETPARAMMENU, (int
*)
NULL);
```

```
    (void) install_var("lrate", Float,(int *) & lrate, 0, 0, NOMENU);
    (void) install_var("crate", Float,(int *) & crate, 0, 0, NOMENU);
    (void) install_var("ecrit", Float,(int *) & ecrit, 0, 0,
SETPCMENU);
    (void) install_var("tmax", Float,(int *) & tmax, 0, 0,
SETPARAMMENU);
}


define_system() {
    register int    i,j, totnum;
    register float *wi, *shwi, *shdwt;
    register float *wt, *shwt, *wtend;
    float *tmp;


    if (!nunits) {
     put_error("cannot init bp system, nunits not defined");
     return(FALSE);
    }
    else
     if (!noutputs) {
         put_error("cannot init bp system, noutputs not defined");
         return(FALSE);
     }
    else
     if (!ninputs) {
         put_error("cannot init bp system, ninputs not defined");
         return(FALSE);
     }
    else

     if (!(nunits && noutputs && ninputs)) {
         put_error("cannot run bp system, nunits not defined");
         return(FALSE);
     }
netinput = &lumped[5*nunits];
    netinput = (float *) emalloc((unsigned)(sizeof(float) * nunits));
    (void) install_var("netinput", Vfloat,(int *) netinput, nunits, 0,

SETSVMENU);
    for (i = 0; i < nunits; i++)
     netinput[i] = 0.0;

    (void) install_var("pss", Float,(int *) &lumped[6*nunits], 0, 0,
SETSVMENU);
pss = 0.0;


    (void) install_var("tss", Float,(int *) &lumped[6*nunits + 1], 0,
0, SETSVMENU);
tss = 0.0;
```

```
    (void) install_var("momentum", Float,(int *) &lumped[6*nunits +
2],0,0,SETPARAMMENU);
momentum = 0.9;

/*   (void) install_var("lrate", Float,(int *) &lumped[6*nunits + 3],

0, 0, NOMENU);
lrate = 0.5;*/


/* activation, bias, target, error, delta arrays lumped here in shared

memory */
/* lumped[] defined in weights because of the order functions are
initialized*/

activation = &lumped[0];
/*    activation = (float *) emalloc((unsigned)(sizeof(float) *
nunits)); */
    (void) install_var("activation",Vfloat,(int
*)activation,nunits,0,SETSVMENU);
/*    for (i = 0; i < nunits; i++)
     activation[i] = 0.0;*/

delta = &lumped[nunits];
/* delta = (float *) emalloc((unsigned)(sizeof(float) * nunits));*/
(void) install_var("delta", Vfloat,(int *) delta, nunits, 0,
SETSVMENU);
/*    for (i = 0; i < nunits; i++)
     delta[i] = 0.0;*/

error = &lumped[2*nunits];
/*   error = (float *) emalloc((unsigned)(sizeof(float) * nunits));*/
    (void) install_var("error", Vfloat,(int *) error, nunits, 0,
SETSVMENU);
/*    for (i = 0; i < nunits; i++)

     error[i] = 0.0;*/

target = &lumped[7*nunits];
/*   target = (float *) emalloc((unsigned)(sizeof(float) *
noutputs));*/
  (void) install_var("target", Vfloat,(int *) target, noutputs,
          0, SETSVMENU);
/*    for (i = 0; i < noutputs; i++)
     target[i] = 0.0;*/

dweight = &wtwed[2*nunits];
/*    dweight = ((float **)
        emalloc((unsigned)(sizeof(float *)*nunits))); */
    (void) install_var("dweight", PVweight,(int *) dweight, nunits,
                              nunits, SETSVMENU);
```

```
totnum = 0;
for(i =0; i<nunits; i++) /*get total number of inputs for "s" variable

of*/
     totnum += num_weights_to[i];    /* wed and weight arrays to
properly
size sh mem segment*/


   dwtssize = (unsigned)(sizeof(float) *(nunits + totnum));
    dwtsid = shmget(0, dwtssize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("dwtsid is %10d\n", dwtsid);
errno = 0;
dwtspage = calc_page(dwtssize);
    dwts = (float *) shmat(dwtsid, dwtspage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

     for(i = 0; i< (nunits +totnum); i++)
     dwts[i] = 0.0;


dbias = &lumped[4*nunits];
/*    dbias = (float *) emalloc((unsigned)(sizeof(float) * nunits));
*/
    (void) install_var("dbias", Vfloat,(int *) dbias,
                              nunits, 0, SETSVMENU);
 /*    for (i = 0; i < nunits; i++)
     dbias[i] = 0.0; */

/* misc array for shmem flags */
errno = 0;
    miscsize = (unsigned)(sizeof(int) * 15);
    miscid = shmget(0, miscsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("miscid is %10d\n", miscid);

errno = 0;
miscpage = calc_page(miscsize);
    misc = (int *) shmat(miscid, miscpage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
    (void) install_var("patno", Int,(int *) & misc[7], 0, 0,
SETSVMENU);

for (i = 0; i <12; i++)
```

```
     misc[i] = 0;
errno = 0;


/*weight "s" array  defined first */

    wtsssize = (unsigned)(sizeof(float) *(nunits +  totnum));
    wtssid = shmget(0, wtsssize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("wtssid is %10d\n", wtssid);
errno = 0;
wtsspage = calc_page(wtsssize);
    wtss = (float *) shmat(wtssid, wtsspage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

/* now wed  "s" array defined*/
    wdssize = (unsigned)(sizeof(float) *(nunits + totnum));
    wdsid = shmget(0, wdssize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("wdsid is %10d\n", wdsid);
errno = 0;
wdspage = calc_page(wdssize);
    wds = (float *) shmat(wdsid, wdspage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

/* now get contents of wed and weight "s" arrays and put them into
shemem wed and wts arrays */
shwt = &wtss[0];
shwi = &wds[0];
for(i = 0; i< nunits; i++){
     wt = weight[i];
     wi = wed[i];
     wtend = wt + num_weights_to[i];
     for(; wt < wtend ;){
     *shwt++ = *wt++;
     *shwi++ = *wi++;
        } /*end wt for*/

}  /*end for*/


/* now attach proper addresses to wed, weight and dweight "r" arrays
to
access repsective "s" arrays*/
```

```
totnum = 0;
for(i = 0; i <nunits; i++) {
     if(num_weights_to[i] !=0){
        weight[i] = &wtss[totnum];
        wed[i] = &wds[totnum];
           dweight[i] = &dwts[totnum];

        totnum += num_weights_to[i];
     } /* end if */
} /* end for */


    System_Defined = TRUE;
    return(TRUE);
}


float  logistic (x)
float  x;
{
    double  exp ();

#ifdef MSDOS
/* we are conservative under msdos to avoid potential underflow
   problems that may arise from returning extremal values -- jlm */
    if (x > 11.5129)
     return(.99999);
      else
     if (x < -11.5129)
         return(.00001);
#else
/* .99999988 is very close to the largest single precis value
   that is resolvably less than 1.0 -- jlm */
      if (x > 16.0)
     return(.99999988);
      else
     if (x < -16.0)
         return(.00000012);
#endif
    else
       return(1.0 / (1.0 + (float) exp( (double) ((-1.0) * x))));
}

init_output() {
    register int i,j;
    register float *sender, *wt, *end;
    float net;

    /* initializes the network to asymptotic outputs given 0 input */

    cycleno = 0;
```

```
    for (i = ninputs; i < nunits; i++) {/* to this unit */
     net = bias[i];
     sender = &activation[first_weight_to[i]];
     wt = weight[i];
     end = sender + num_weights_to[i];
     for (j = first_weight_to[i]; j < ninputs; j++) {
           sender++; wt++; /* step over input units to
                           initialize to all-zero input case */
     }
     for (; sender < end ; ){/* from this unit */
         net += (*sender++) * (*wt++);
     }
     netinput[i] = net;
     lumped[i] = activation[i] = (float) logistic(net);
    }
    if (step_size < PATTERN) {
     update_display();
     if (single_flag) {
         if (contin_test() == BREAK) return (BREAK);
     }
    }
    if (Interrupt) {
         Interrupt_flag = 0;
         update_display();
         if (contin_test() == BREAK) return (BREAK);
    }
    return(CONTINUE);
}

cycle() {
    register int i,cy;
    register float *sender,*wt,*end;
    float newinput;

    for (cy = 0; cy < ncycles; cy++) {
     cycleno++;
     for (i = ninputs; i < nunits; i++) {/* to this unit */
         newinput = bias[i];
         sender = &activation[first_weight_to[i]];
         end = sender + num_weights_to[i];
         wt = weight[i];
         for (;sender<end;) {/* from this unit */
           newinput += (*sender++) * (*wt++);
         }
         netinput[i] = crate * newinput + drate * netinput[i];
         activation[i] = (float) logistic(netinput[i]);
     }
     if (step_size == CYCLE) {
         update_display();
         if (single_flag) {
           if (contin_test() == BREAK) return (BREAK);
         }
```

```
        }
        if (Interrupt) {
            update_display();
            Interrupt_flag = 0;
            if (contin_test() == BREAK) return (BREAK);
        }
       }
       if (step_size == NCYCLES) {
        update_display();
       }
       return(CONTINUE);
}


compute_output() {
    register int    i;
    float *sender, *wt, *end;
    float net;
    compout2flag = 0;
    for (i = ninputs; i < nunits; i++) {  /* to this unit */
     net = bias[i];
     sender = &activation[first_weight_to[i]];
        end = sender + num_weights_to[i];
     wt = weight[i];
     for (; sender < end ;)
         net += (*sender++)*(*wt++); /* from this unit */
     netinput[i] = net;
activation[i] =(float) (1.0 / (1.0 + (float) exp( (double) ((-1.0) *
net))));
    }
     compout2flag = 1;
}

compute_error() {
    register int i,j;
    float *wt, *sender, *end;
    float del;
comperrorflag1 = 0;
comperrorflag2 = 0;

for (i = ninputs; i < nunits - noutputs; i++) {
     error[i] = 0.0;
}
while(!compout2flag);
compout2flag = 0;
for (i=nunits-noutputs, j=0; i < nunits ; j++, i++){
     if(target[j] >= 0)  /*  We care about this one*/
        error[i]= target[j] - activation[i];
     else
         error[i] = 0.0;
 }
comperrorflag1 = 1;
```

```
   for (i= nunits - 1; i >= ninputs; i--) {
     del  =  delta[i]  =  error[i]  *  activation[i]  *  (1.0  -
activation[i]);

     if (first_weight_to[i] + num_weights_to[i] < ninputs) continue;
     /* no point in propagating error back to input units */
     sender = &error[first_weight_to[i]];
     end = sender +    num_weights_to[i];
     wt = weight[i];
     for (;sender < end;) {
         *sender++ += del * (*wt++);
     }
    }
comperrorflag2 = 1;
}

compute_wed() {
    register int   i;
    float *wi, *sender, *end;
    float del;
    compwedflag2 = 0;
while (!comperrorflag2);
comperrorflag2 = 0;
   for (i = ninputs; i < nunits; i++) {
     sender = &activation[first_weight_to[i]];
     end = sender + num_weights_to[i];
     del = delta[i];
     wi = wed[i];
     for (;sender < end;)
         *wi++ += del * (*sender++);

     if( i <=(nunits - 1)) compwedflag1 = 1;
  }
compwedflag2 = 1;
}

clear_wed() {
    register int   i,j,num;
    register float *wi, *end;

    for (i = ninputs; i < nunits; i++) {
     bed[i] = 0.0;
     wi = wed[i];
     end = wi + num_weights_to[i];
     for (; wi < end;) {
         *wi++ = 0.0;
     }
    }
}

change_weights() {
    register int    i;
```

```
    register float *wt, *dwt, *epi, *wi, *end;

/*while(!compwedflag2);
compwedflag2 = 0; */
/*    link_sum();*/
  for (i = ninputs; i < nunits; i++) {

while(!compwedflag1 && !compwedflag2);
     compwedflag1 = 0;
     dbias[i] = lrate*delta[i]+ momentum * dbias[i]; /*lrate vs
bepsilon
                          delta[i]*/
     bias[i] += dbias[i];
     wt = weight[i];
     dwt= dweight[i];
     wi = wed[i];
     end = wt + num_weights_to[i];
     for (; wt < end; ) {
         *dwt = lrate * (*wi) + momentum * (*dwt); /*lrate vs (*epi++)

*/
         *wt++ += *dwt++;
         *wi++ = 0.0;
     }
    }
  /*  pos_neg_constraints();*/
}

float p_css = (float) 0.0;
float css = (float) 0.0;

change_weights_follow() {
    register int    i;
    register float *wt, *dwt, *epi, *wi, *end, *pwi;
    float tb, dp, den;

    p_css = css;
    css = 0.0;
    dp = 0.0;

    link_sum();

    for (i = ninputs; i < nunits; i++) {
        tb = bed[i];
     dbias[i] = tb*bepsilon[i] + momentum * dbias[i];
     bias[i] += dbias[i];
     css += ((double) tb)*((double) tb);
     dp += ((double) tb)*((double) pbed[i]);
     pbed[i] = tb;
     bed[i] = 0.0;
     wt = weight[i];
     dwt= dweight[i];
```

```
        wi = wed[i];
        pwi = pwed[i];
        epi = epsilon[i];
        end = wt + num_weights_to[i];
        for (; wt < end; ) {
            *dwt = (*epi++)*(*wi) + momentum * (*dwt);
            *wt++ += *dwt++;
            css += ((double) (*wi))*((double) (*wi));
            dp += ((double) (*wi))*((double) (*pwi));
            *pwi++ = *wi;

            *wi++ = 0.0;
        }
    }

    den = p_css * css;
    if (den > 0.0) gcor = dp/(sqrt(den));
    else gcor = 0.0;

    pos_neg_constraints();
}

constrain_weights() {
    pos_neg_constraints();
    link_constraints();
}

pos_neg_constraints() {
    float **fpt;

    for (fpt = positive_constraints; fpt && *fpt; fpt++)
     if (**fpt < 0.0)
         **fpt = 0.0;

    for (fpt = negative_constraints; fpt && *fpt; fpt++)
     if (**fpt > 0.0)
         **fpt = 0.0;
}

link_constraints() {
    register int    i,j;
    float   t;

    for (i = 0; i < nlinks; i++) {
     t = *constraints[i].cvec[0];
     for (j = 1; j < constraints[i].num; j++) {
         *constraints[i].cvec[j] = t;
     }
    }
}

link_sum() {
```

```
    register int    i,j;
    float   ss;

    for (i = 0; i < nlinks; i++) {
     ss = 0.0;
     for (j = 0; j < constraints[i].num; j++) {
         ss += *constraints[i].ivec[j];
     }
     for (j = 0; j < constraints[i].num; j++) {
         *constraints[i].ivec[j] = ss;
     }
    }
}


setinput() {
    register int    i,prev_index;
    register float  *pp;

    for (i = 0, pp = ipattern[patno]; i < ninputs; i++, pp++) {
         activation[i] =  *pp;
    }

    strcpy(cpname,pname[patno]);
}

settarget() {
    register int    i;
    register float *pp;

    for (i = 0, pp = tpattern[patno]; i < noutputs; i++, pp++) {
     target[i] = *pp;
     if (target[i] == 1.0) {
         target[i] = tmax;
     }
     else if(target[i] == 0.0) {
             target[i] = 1 - tmax;
     }
    }
}

setup_pattern() {
    setinput();
    settarget();
}
tallcompute_error() {
    register int i, j;
    float *wt, *sender, *end;
    float del;

for (i = ninputs; i < nunits - noutputs; i++) {
     error[i] = 0.0;
```

```
    }
for (i=nunits-noutputs, j = 0; i < nunits ; j++, i++){
     if(target[j] >= 0)  /*  We care about this one*/
        error[i]= target[j] - activation[i];
     else
         error[i] = 0.0;
 }
   for (i= nunits - 1; i >= ninputs; i--) {
     del =delta[i] = error[i]* activation[i] * (1.0 - activation[i]);
     if (first_weight_to[i] + num_weights_to[i] < ninputs) continue;
     /* no point in propagating error back to input units */
     sender = &error[first_weight_to[i]];
     end = sender +    num_weights_to[i];
     wt = weight[i];
     for (;sender < end;) {
         *sender++ += del * (*wt++);

     }
    }
}

talltrial() {

 setup_pattern();
    if (cascade) {
     if (init_output() == BREAK) return (BREAK);
     if (cycle() == BREAK) return (BREAK);
    }
    else  {
        compute_output();
        if (step_size < PATTERN) {
           update_display();
           if (single_flag) {
               if (contin_test() == BREAK) return(BREAK);
           } /* end single */
     } /* end stepsize */
} /* end else */
    tallcompute_error();
comperrorflag1 = 1;
   sumstats();
    return (CONTINUE);
} /* end of talltrial*/

trial() {

if (!forkflag){
     forkflag = 1;
     errno = 0;
     pid = fork();
     if (errno >0)
fprintf(fp, "Trouble with fork, errno =%3d:  %s\n", errno,
sys_errlist[errno]);
```

```
}
if(pid == 0){
errno = 0;
function = mpadvise(MPA_CPU_SET, 4);
if(errno > 0)
fprintf(stderr, "At 2nd proc call errno is %3d, which means %s\n",
errno, sys_errlist[errno]);
errno = 0;
forkagain:
if(doneflag){
     forkflag = 0;
     exit(0);
     }
compute_error();

change_weights();
while(!parentflag);
parentflag = 0;

goto forkagain;

} /*end of pid eq 0 */
if(pid != 0) return;
} /* end of train */

sumstats() {
    register int    i,j;
    register float t;
    pss = 0.0;

while(!comperrorflag1);
comperrorflag1 = 0;
    for (j = 0,i =nunits - noutputs; i <nunits; i++,j++) {
      if (target[j] >= 0) {
           t = error[i];
     pss += t*t;
      }
    }
    tss += pss;
}
ptrain() {
  return(train('p'));
}

strain() {
  return(train('s'));
}

train(c) char c; {
    int     t,i,old,npat;
    char    *str;
    parentflag = 0;
```

```
    doneflag = 0;
    forkflag = 0;

    if (!System_Defined)
     if (!define_system())
         return(BREAK);

    /* in case prev epoch was terminated early we clear the weds and
beds */
    clear_wed();
    cycleno = 0;
    for (t = 0; t < nepochs; t++) {
      epochno++;
     for (i = 0; i < npatterns; i++)
         used[i] = i;
     if (c == 'p') {
       for (i = 0; i < npatterns; i++) {
         npat = rnd() * (npatterns - i) + i;
         old = used[i];
         used[i] = used[npat];
         used[npat] = old;

       }
     }
     tss = 0.0;

     for (i = 0; i < npatterns; i++) {
       patno = used[i];
     if(!forkflag){
     setinput();
     settarget();
        trial();
     } /* end of forkflag */

compute_output();
sumstats();
if(lflag) {
compute_wed();
     if (i != (npatterns -1)) {
           patno = used[i+1];
           setinput();
           settarget();
           parentflag = 1;
     } /* end of i not eq last pattern */

} /* end of lflag */
/*   if (step_size == PATTERN) {
      update_display();
      if (single_flag) {
               if (contin_test() == BREAK) return(BREAK);
           } end of single flag
      }  end of PATTERN
```

```
*/

} /* end of npatterns for loop*/
if (tss < ecrit) break;
     if (t!= (nepochs - 1)){
     patno = used[0];
     setinput();
     settarget();
     parentflag = 1;
     } /* end of t ne loop */

         if (Interrupt) {
           Interrupt_flag = 0;
           update_display();
           if (contin_test() == BREAK) return(BREAK);
         }

     if (step_size == EPOCH) {   /* defined as 4*/
      update_display();
      if (single_flag) {
               if (contin_test() == BREAK) return(BREAK);
         } /*end of single flag*/
      } /* end of EPOCH */
}  /* end of nepochs for loop */


 doneflag = 1;
 parentflag = 1;
if (step_size == NEPOCHS) {   /* defined as 5 */
      update_display();
    }
    return(CONTINUE);
}

talltrain() {
    int     t,i,old,npat;
    char    *str;

    if (!System_Defined)
     if (!define_system())
         return(BREAK);
    /* in case prev epoch was terminated early we clear the weds and
beds */
       cycleno = 0;
     tss = 0.0;
     for (i = 0; i < npatterns; i++) {
       patno = used[i] = i;
           if(talltrial() == BREAK) return(BREAK);

         if (step_size == PATTERN) {
           update_display();
           if (single_flag) {
```

```
                if (contin_test() == BREAK) return(BREAK);
              }
            }
            if (Interrupt) {
              Interrupt_flag = 0;
              update_display();
              if (contin_test() == BREAK) return(BREAK);
            }
        } /* end of npatterns for loop*/

return(CONTINUE);
} /* end of talltrain */

tall() {
  int save_lflag;
  int save_single_flag;
  int save_nepochs;
  int save_step_size;

  save_lflag = lflag;  lflag = 0;
  save_single_flag = single_flag;
  if (in_stream == stdin) single_flag = 1;
  save_step_size = step_size;
  if (step_size > PATTERN) step_size = PATTERN;
  save_nepochs = nepochs;  nepochs = 1;
  tallflag = 1;
  talltrain();

  tallflag = 0;
  lflag = save_lflag;
  nepochs = save_nepochs;
  single_flag = save_single_flag;
  step_size = save_step_size;
  return(CONTINUE);
}

test_pattern() {
    char   *str;
    int save_single_flag;
    int save_step_size;

    if (!System_Defined)
     if (!define_system())
         return(BREAK);

    tss = 0.0;

    str = get_command("Test which pattern? ");
    if(str == NULL) return(CONTINUE);
    if ((patno = get_pattern_number(str)) < 0) {
       return(put_error("Invalid pattern specification."));
    }
```

```
    if (cascade) {
        save_single_flag = single_flag; single_flag = 1;
     save_step_size = step_size; step_size = CYCLE;
    }
    talltrial();
    update_display();
    if (cascade) {
     single_flag = save_single_flag;
     step_size = save_step_size;
    }
    return(CONTINUE);
}

newstart() {
     random_seed = rand();
     reset_weights();
}

reset_weights() {
    register int    i,j,k,first,num;
    char ch;

    epochno = 0;
    pss = tss = gcor = 0.0;
    cpname[0] = '\0';
    srand(random_seed);

    if (!System_Defined)
     if (!define_system())
         return(BREAK);


    for (j = 0, k = 3*nunits; j < nunits; j++, k++) {
     first = first_weight_to[j];
     num = num_weights_to[j];
      for (i = 0; i < num; i++) {
     wed[j][i] = dweight[j][i] = 0.0;
     if (pwed) pwed[j][i] = 0.0;
     ch = wchar[j][i];
     if (isupper(ch)) ch = tolower(ch);
     if (ch == '.') {
         weight[j][i] = 0.0;
     }
     else {
         if (constants[ch - 'a'].random) {
               if (constants[ch - 'a'].positive) {
                 weight[j][i] = wrange * rnd();
               }
               else
                 if (constants[ch - 'a'].negative) {
                     weight[j][i] = wrange * (rnd() - 1);
                 }
```

```
                else
                  weight[j][i] = wrange * (rnd() -.5);
            }
            else {
                weight[j][i] = constants[ch - 'a'].value;
            }
        }
         }
         bed[j] = dbias[j] = 0.0;
         if (pbed) pbed[j] = 0.0;
         ch = bchar[j];
         if (isupper(ch)) ch = tolower(ch);
         if (ch == '.') {
            bias[j] = 0;
         }
         else {
            if (constants[ch - 'a'].random) {
                if (constants[ch - 'a'].positive) {
                  bias[j] = wrange * rnd();
                }
                else
                  if (constants[ch - 'a'].negative) {
                      bias[j] = wrange * (rnd() - 1);
                  }
                else
                  bias[j]  = wrange * (rnd() -.5);
            }
            else {
                bias[j]  = constants[ch - 'a'].value;
            }
          }
        }
        constrain_weights();

        for (i = 0, j = 4*nunits; i < noutputs; i++, j++) {
          lumped[j] = target[i] = 0.0;
        }
        for (i = 0; i < nunits; i++)
          netinput[i] = activation[i] = delta[i] = error[i] = 0.0;

        for (i = 0; i < 3*nunits; i++)
         lumped[i] = 0.0;

        update_display();
        return(CONTINUE);
}

set_lgrain() {
        char old_grain_string[STRINGLENGTH];
        struct Variable *vp, *lookup_var();

        strcpy(old_grain_string,grain_string);
```

```
    vp = lookup_var("lgrain");
    change_variable("lgrain",vp);

    if(startsame(grain_string,"epoch"))strcpy(grain_string,"epoch");
    else if (startsame(grain_string,"pattern"))
                              strcpy(grain_string,"pattern");
    else {
     strcpy(grain_string,old_grain_string);
     return(put_error("urecognized grain -- not changed."));
    }
    return(CONTINUE);
}

set_follow_mode() {
    struct Variable *vp, *lookup_var();
    int pv, i, j;
    pv = follow;

    vp = lookup_var("follow");
    change_variable("follow",vp);

    if (follow == 0) return (CONTINUE);
    if (pwed == NULL) {
      pwed = ((float **) emalloc((unsigned)(sizeof(float *)*nunits)));
      (void) install_var("pwed", PVweight,(int *) pwed, nunits,
                              nunits, NOMENU);
      for (i = 0; i < nunits; i++) {
     pwed[i] = ((float *)
           emalloc((unsigned)(sizeof(float)*num_weights_to[i])));
      }

      pbed = ((float *) emalloc((unsigned)(sizeof(float) * nunits)));
      (void) install_var("pbed", Vfloat,(int *) pbed,
                              nunits, 0, NOMENU);
    }

    if (pv == 0) {
      for (i = 0; i < nunits; i++) {
     for (j = 0; j < num_weights_to[i]; j++) {
         pwed[i][j] = 0.0;
     }
      }
      for (i = 0; i < nunits; i++)
     pbed[i] = 0.0;
    }
    gcor = css = 0.0;
    return(CONTINUE);
}

change_crate() {
    struct Variable *varp;
```

```
    if ((varp = lookup_var("crate")) != NULL) {
     change_variable("crate",(int *) varp)
    }
    else {
     return(put_error("crate is not defined"));
    }
    drate = 1 - crate;
    return(CONTINUE);
}

init_weights() {
    int define_bp_network();
    (void) install_command("network", define_bp_network,GETMENU,(int
*)
NULL);
    (void) install_command("weights", read_weights, GETMENU,(int *)
NULL);
    (void) install_command("weights", write_weights, SAVEMENU,(int *)
NULL);
    (void) install_var("nunits", Int,(int *) & nunits, 0, 0,
SETCONFMENU);
    (void) install_var("ninputs", Int,(int *) & ninputs, 0, 0,
SETCONFMENU);
    (void) install_var("noutputs", Int,(int *) & noutputs, 0, 0,
SETCONFMENU);
    (void) install_var("wrange",Float,(int *) &wrange,0,0,
SETPARAMMENU);
}
```

WTSH.C

```
/* file: weights.c MODIFIED FOR HYBRID TO wtsh.c

     read in network descriptions, and set up constraints.
     First version implemented by Elliot Jaffe.
     Date of last revision: 8-12-87/JLM.

     HYBRID MODIFIED  Masscomp revisions */


/* the following is the form for network description files.

definitions:
nunits <int>
ninputs <int>
noutputs <int>
maxconstraints <int>
constraints:
<char> <float> or <char> [random positive negative linked]
...
end
```

```
network:
<strings of . and chars as defined in definitions:>
end
biases:
<a single line of . and chars as biases for units>
end
sigmas:
<a single line of .'s and chars specifying sigmas -- harmony theory
only>
end
<EOF>
*/


#include "general.h"
#include "command.h"
#include "wtsh.h"
#include "variable.h"

float **weight = NULL;
char   **wchar;              /* pointers to vectors of chars
                                that are used in resetting weights*/
float  *bias = NULL;
char   *bchar;                 /* like wchar */
float **epsilon;
float  *bepsilon = NULL;     /* thresh epsilon array */
float  **wed = NULL;
float  *bed = NULL;
float  *sigma = NULL;  /* strength parameter for knowledge atoms */

struct constants    constants[26];

float **positive_constraints;
float **negative_constraints;
 /* Array of struct constraint, for keeping links together */
struct constraint  *constraints = NULL;

float   lrate = 0.5;

float   wrange = 1;

int     nunits = 0;

int     ninputs = 0;
int     noutputs = 0;
int  maxpos = MAXCONSTRAINTS;
int  maxneg = MAXCONSTRAINTS;
int     nlinks = 0;
static  nposconstr = 0;
static  nnegconstr = 0;
int     epsilon_menu = SETWTMENU;
char    net_descr_name[BUFSIZ];
```

```
int lumpid, lumppage;  /* added for shared mem 11 apr 89 bb */
unsigned lumpsize;
float *lumped = NULL;

int fnwtid, fnwtpage;
unsigned fnwtsize;
int *fnwt = NULL;

int wtwedid, wtwedpage;
unsigned wtwedsize;
float **wtwed = NULL;

int bp;    /* TRUE if program is bp */

# define ENLARGE_POS -1
# define ENLARGE_NEG -2

define_bp_network() {
     bp = 1;
     define_net();
}

define_network() {
     bp = 0;
     define_net();
}

define_net() {
    char   *sp;
    char    string[BUFSIZ];
    FILE * sv_instream;
    struct Variable *lookup_var ();
    int     i;
    boolean defined_weights = FALSE;

    sv_instream = in_stream;
    sp = get_command("filename for network description: ");
    if ( sp == NULL) return(CONTINUE);
    strcpy(net_descr_name,sp);

    if ((in_stream = fopen(sp, "r")) == NULL) {
     in_stream = sv_instream;
     return(put_error("Can't open network file."));
    }


    nlinks = 0;

    for (i = 0; i < 26; i++) {
     constants[i].random = FALSE;
     constants[i].positive = FALSE;
     constants[i].negative = FALSE;
```

```
 constants[i].link = FALSE;
 constants[i].value = 0.0;
}

constants['r' - 'a'].random = TRUE;
constants['p' - 'a'].random = TRUE;
constants['p' - 'a'].positive = TRUE;
constants['n' - 'a'].random = TRUE;
constants['n' - 'a'].negative = TRUE;

while (fscanf(in_stream, "%s", string) != EOF) {
 if (!strcmp(string, "definitions:")) {
     if (read_definitions() == BREAK) {
         fclose(in_stream); in_stream = sv_instream;
       return(BREAK);
     }
 }
 else
     if (!strcmp(string, "constraints:")) {
       if (read_constraints(constants) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
       }
     }
 else
     if (!strcmp(string, "network:")) {
       defined_weights = read_network(constants);
       if (!defined_weights) {
        if (put_error(err_string) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
        }
       }
     }
 else
     if (!strcmp(string, "biases:")) {
       if (read_biases(constants) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
       }
     }
 else
     if (!strcmp(string, "sigmas:")) {
       if (read_sigmas(constants) == BREAK) {
             fclose(in_stream); in_stream = sv_instream;
             return(BREAK);
       }

     }
 else
     if (!strcmp(string, "end")) {
     /* just skip over it */
```

```
        }
      else {
          sprintf(err_string,
            "error reading network file: I don't understand
%s\n",string);
          if (put_error(err_string) == BREAK) {
            fclose(in_stream); in_stream = sv_instream;
            return(BREAK);
          }
      }
    }
    fclose(in_stream);
    in_stream = sv_instream;
    if (nlinks)
     constrain_weights();
    return(CONTINUE);
}

read_definitions() {
    char    string[BUFSIZ];
    struct Variable *varp,
                    *lookup_var ();

    while (fscanf(in_stream, "%s", string) != EOF) {
     if (!strcmp(string, "end"))
         return(CONTINUE);
     if ((varp = lookup_var(string)) != NULL) {
         change_variable(string,(int *) varp);
     }
     else {
         sprintf(err_string,
           "Error: unknown variable in network file, %s\n", string);
         return(put_error(err_string));
     }
    }
}

read_network(con)
struct constants   *con;
{
    int     i,r,s,block,since_first,last_weight_to,tempint;
    int        rstart,rnum,rend,sstart,snum,send,con_index;
    char    ch,all_ch,*strp;
    char    string[BUFSIZ];
    int        needline = 1;
    float   *tmp; char *ctmp;

    (void) srand(random_seed);


/* activation, bias, target, error, delta arrays lumped here in shared
```

```
memory */
errno = 0;
    lumpsize = (unsigned)(sizeof(float) *(7*nunits + noutputs));
    lumpid = shmget(0, lumpsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("lumpid is %10d\n", lumpid);
errno = 0;
lumppage = calc_page(lumpsize);
    lumped = (float *) shmat(lumpid, lumppage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
for (i = 0; i <6*nunits + noutputs; i++)
     lumped[i] = 0.0;
errno = 0;

/* weight and wed "r" arrays for shared memory  */
    wtwedsize = (unsigned)(sizeof(float *) *(3*nunits));
    wtwedid = shmget(0, wtwedsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("wtwedid is %10d\n", wtwedid);
errno = 0;
wtwedpage = calc_page(wtwedsize);
wtwed = (float **) shmat(wtwedid, wtwedpage, 0);
if (errno >0)
fprintf(stderr, "%s\n", sys_errlist[errno]);
errno = 0;

weight = &wtwed[0];
/*   weight = ((float**) emalloc((unsigned int)(sizeof(float *) *
nunits))); */

    epsilon = ((float **) emalloc((unsigned int)(sizeof(float *)) *
nunits));

    wchar = ((char **) emalloc((unsigned int)(sizeof(char *) *
nunits)));

fnwtsize = (unsigned)(sizeof(int *) *(2*nunits));
    fnwtid = shmget(0, fnwtsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "errno is %3d,  which means %s\n", errno,
sys_errlist[errno]);
printf("fnwtid is %10d\n", fnwtid);
errno = 0;
fnwtpage = calc_page(fnwtsize);
fnwt = (int *) shmat(fnwtid, fnwtpage, 0);

if (errno >0)
```

```
fprintf(stderr, "%s\n", sys_errlist[errno]);

errno = 0;

first_weight_to = &fnwt[0];
/*first_weight_to = (int *) emalloc((unsigned int)(sizeof(int) *
nunits)); */
/*    for (r = 0; r < nunits; r++)
     first_weight_to[r] = nunits;
*/
num_weights_to = &fnwt[nunits];
 /*num_weights_to = (int *) emalloc((unsigned int)(sizeof(int) *
nunits)); */


    for (r = 0; r < nunits; r++){
     num_weights_to[r] = 0;
        first_weight_to[r] = nunits;
    }


   (void) install_var("weight",PVweight,(int *) weight,nunits,nunits,
                                          SETWTMENU);
    (void) install_var("epsilon", PVweight,(int *) epsilon, nunits,
nunits,
                                              epsilon_menu);
    if (bp) {
wed = &wtwed[nunits];
/*     wed = ((float**) emalloc((unsigned int)(sizeof(float *) *
nunits))); */
     (void) install_var("wed",PVweight,(int *) wed,nunits,nunits,
                                          SETSVMENU);
    }

    rstart = 0; rend = nunits -1; sstart = 0; send = nunits -1;
    for (block = 0; ; block++) {
gbagain:
      if (fscanf(in_stream,"%s",string) == EOF) {
         sprintf(err_string,"error in network description");
         return(FALSE);
      }
      if (strcmp("end",string) == 0) {
           if (block) return(TRUE);
     else {
      sprintf(err_string,"error in network description");
     }
     return(FALSE);
      }
      all_ch = '\0';
      if (string[0] == '%') {
        fscanf(in_stream,"%d%d%d%d",&rstart,&rnum,&sstart,&snum);
     rend = rstart + rnum -1;
```

```
      send = sstart + snum -1;
      if (string[1]) {
          all_ch = string[1];
      }

       }
       else {
        if (!block) {
      needline = 0;
        }
        else {
          sprintf(err_string,"error in network description");
          return(FALSE);
        }
       }
       for (r = rstart; r <= rend; r++) {
        if (!all_ch) {
          if (needline) {
              if (fscanf(in_stream,"%s",string) == EOF) {
             sprintf(err_string,"not enough units in network
description");
             return(FALSE);
           }
          }
          else needline = 1;
        }
        else {
          for (s = 0; s < snum; s++) string[s] = all_ch;
          string[s] = '\0';
        }
        first_weight_to[r] = sstart;
        last_weight_to = send;
        num_weights_to[r] = 1 + last_weight_to - first_weight_to[r];
  weight[r] = ((float *)
         emalloc ((unsigned int)(sizeof(float) * num_weights_to[r])));

  epsilon[r] = ((float *)
            emalloc    ((unsigned    int)(sizeof(float)    *
num_weights_to[r])));

  wchar[r] = ((char *)
             emalloc    ((unsigned    int)(sizeof(char)    *
num_weights_to[r])));
        if (bp) {
  wed[r] = ((float *)
     emalloc ((unsigned int)(sizeof(float) * num_weights_to[r])));

        }
        for(s = 0; s < num_weights_to[r]; s++) {
           weight[r][s] = 0.0;
           epsilon[r][s] = 0.0;
           wchar[r][s] = '.';
```

```
        if (bp) wed[r][s] = 0.0;
      }
      for (strp = string,s = sstart,since_first = 0; s <= send; s++)
{
                        /* loop over the from units */
        ch = *strp++;
        wchar[r][since_first] = ch;
        if (ch == '.') {
          since_first++;
        }

        else {
        /* first check if this is realy a character */
          if (!isalpha(ch)) {
              sprintf(err_string,"non_alpha character in network");
              return(FALSE);
          }


        /* upper case means this weight is non-changable */
          if (isupper(ch)) {
          /* make it lower case */
              ch = tolower(ch);
              epsilon[r][since_first] = 0;
          }
          else {
              epsilon[r][since_first] = lrate;
          }

        /* now set up the char based on the stored con definitions */
          if (con[ch - 'a'].random) {
              if (con[ch - 'a'].positive) {
                if (nposconstr >= maxpos) {
                    enlarge_constraints(ENLARGE_POS);
                }
                weight[r][since_first] = wrange * rnd();
                positive_constraints[nposconstr++] =
                     &weight[r][since_first];
              }
              else
                if (con[ch - 'a'].negative) {
                    if (nnegconstr >= maxneg){
                      enlarge_constraints(ENLARGE_NEG);
                    }
                    weight[r][since_first] =
                         wrange * (rnd() - 1);
                    negative_constraints[nnegconstr++] =
                         &weight[r][since_first];
                }
              else
                weight[r][since_first] = wrange * (rnd() -.5);
          }
```

```
            else {
                weight[r][since_first] = con[ch - 'a'].value;
            }
            if (con[ch - 'a'].link) {
                con_index = (con[ch - 'a'].link - 1);
                if (constraints[con_index].num >=
constraints[con_index].max) {
                      enlarge_constraints(con_index);
                }

                tempint = constraints[con_index].num;
                constraints[con_index].cvec[tempint]
                      = &weight[r][since_first];


                if (bp) {
                  constraints[con_index].ivec[tempint]
                      = &wed[r][since_first];
                }

                  tempint = constraints[con_index].num + 1;
                constraints[con_index].num = tempint;
                /* this kludge (tempint) is for the MS compiler */
            }
          since_first++;
          }
        }
      }
    }
}

read_biases(con)
struct constants   *con;
{
    int     j,rstart,rend,rnum,block,con_index,tempint;
    char    ch,all_ch,*strp;
    char    string[BUFSIZ];

bias = &lumped[3*nunits];
/*bias = (float *) emalloc((unsigned int)(sizeof(float) * nunits));*/
    (void) install_var("bias", Vfloat,(int *) bias, nunits, 0,
SETWTMENU);

    bepsilon = (float *) emalloc((unsigned int)(sizeof(float) *
nunits));
    (void) install_var("bepsilon", Vfloat,(int *) bepsilon, nunits, 0,

                                        epsilon_menu);
    bchar = (char *) emalloc((unsigned int)(sizeof(char) * nunits));

    if (bp) {
     bed = (float *) emalloc((unsigned int)(sizeof(float) * nunits));
```

```
        (void) install_var("bed", Vfloat,(int *)
bed,nunits,0,SETSVMENU);
    }

    for (j = 0; j < nunits; j++) {
     bias[j] = 0.0;
     bepsilon[j] = 0;
     bchar[j] = '.';
     if (bp) bed[j] = 0.0;
    }

    rstart = 0; rend = nunits -1;
  for (block = 0; ; block++) {
gtagain:
    if (fscanf(in_stream,"%s",string) == EOF) {
     return(put_error("problem in bias description"));
    }

    if (strcmp(string,"end") == 0) {
        if (block) return (CONTINUE);
        else return(put_error("problem in bias description"));
    }
    if (string[0] == '%') {
        fscanf(in_stream,"%d%d",&rstart,&rnum);
     rend = rstart + rnum -1;
     if (string[1] != '\0') {
         all_ch = string[1];
         for (j = 0; j < rnum; j++) {
           string[j] = all_ch;
         }
         string[j] = '\0';
     }
     else goto gtagain;
    }
    for (strp = string, j = rstart; j <= rend; j++, strp++) {
     ch = *strp;
     bchar[j] = ch;
     if (ch == '.') {
         bias[j] = 0;
         bepsilon[j] = 0;
     }
     else {
     /* first check if this is realy a character */
         if (!isalpha(ch)) {
           return(put_error("non_alpha character in bias"));
         }

     /* upper case means this weight is non-changable */
         if (isupper(ch)) {
         /* make it lower case */
           ch = tolower(ch);
           bepsilon[j] = 0;
```

```
            }
            else {
              bepsilon[j] = lrate;
            }

            /* now set up the char based on the stored con definitions */
            if (con[ch - 'a'].random) {
                  if (con[ch - 'a'].positive) {
                    bias[j] = wrange * rnd();
                    if (nposconstr >= maxpos) {
                        enlarge_constraints(ENLARGE_POS);
                    }
                    positive_constraints[nposconstr++] = &bias[j];
                  }
                  else
                    if (con[ch - 'a'].negative) {
                        bias[j] = wrange * (rnd() - 1);
                        if (nnegconstr >= maxneg){
                          enlarge_constraints(ENLARGE_NEG);
                        }
                        negative_constraints[nnegconstr++] = &bias[j];
                    }
                  else
                    bias[j] = wrange * (rnd() -.5);
            }
            else {
                  bias[j] = con[ch - 'a'].value;
            }
            if (con[ch - 'a'].link) {
              con_index = (con[ch - 'a'].link - 1);
              if (constraints[con_index].num >=
constraints[con_index].max){
                  enlarge_constraints(con_index);
                }
              tempint = constraints[con_index].num;
              constraints[con_index].cvec[tempint] = &bias[j];
              if (bp) constraints[con_index].ivec[tempint] = &bed[j];
              constraints[con_index].num++;
            }
         }
        }
      }
    }

read_sigmas(con) struct constants   *con; {
    int     j;
    char    ch, all_ch, *strp;
    char    string[BUFSIZ];
    int         rstart, rend, rnum, block;

    sigma = (float *) emalloc((unsigned int)(sizeof(float) * nunits));
    for (j = 0; j < nunits; j++) {
```

```
      sigma[j] = 1.0;          /* default sigma is 1.0 */
    }
    (void) install_var("sigma", Vfloat,(int *) sigma, nunits, 0,
                   SETWTMENU);
    rstart = 0; rend = nunits -1;
  for (block = 0; ; block++) {
gsagain:
    if (fscanf(in_stream, "%s", string) == EOF) {
     return(put_error("problem in sigma description"));
    }
    if (strcmp(string,"end") == 0) {
        if (block) return (CONTINUE);
        else return(put_error("problem in sigma description"));
    }
    if (string[0] == '%') {
        fscanf(in_stream,"%d%d",&rstart,&rnum);
     rend = rstart + rnum -1;
     if (string[1] != '\0') {
         all_ch = string[1];
         for (j = 0; j < rnum; j++) {
           string[j] = all_ch;
         }

         string[j] = '\0';
     }
     else goto gsagain;
    }
    for (strp = string, j = rstart; j <= rend; j++, strp++) {
     ch = *strp;
     if (ch == '.') {
       sigma[j] = 1.0;
     }
     else {
     /* first check if this is really a character */
         if (!isalpha(ch)) {
           return(put_error("non_alpha character in bias"));
         }
         if (isupper(ch)) {
         /* make it lower case */
           ch = tolower(ch);
         }
         sigma[j] = con[ch - 'a'].value;
         if (sigma[j] < 0) {
           return(put_error("can't set sigma less than 0!"));
         }
     }
    }
  }
}

read_constraints(con)
struct constants   *con;
```

```
{
    char    ch;
    float   flt;
    int     isflt;
    char    string[BUFSIZ];
    char    str[5][30];
    int     i,j,ch_ind;
    int        nstr;

    while (fgets(string, BUFSIZ, in_stream) != NULL) {
     if (string[0] == NULL || string[0] == '\n') {
         if (fgets(string, BUFSIZ, in_stream) == NULL) {
           break;
         }
     }
     if (strncmp(string,"end",3) == 0) break;

     ch = '\0';

     for (i = 0; i < 5; i++) str[i][0] = '\0';

     (void) sscanf(string, "%c %s %s %s %s %s",
                  &ch, str[0], str[1], str[2], str[3], str[4]);
     ch = (isupper(ch)) ? tolower(ch) : ch;
     ch_ind = ch - 'a';

     con[ch_ind].random = con[ch_ind].positive =
         con[ch_ind].negative = con[ch_ind].link = FALSE;
     con[ch_ind].value = 0.0;
     for (i = 0; (i < 5) && (str[i][0] != '\0'); i++) {
         if ( (isflt = sscanf(str[i],"%f",&flt)) == 1) {
           con[ch_ind].value = flt;
         }
         else
           if (startsame(str[i], "random"))
               con[ch_ind].random = TRUE;
         else
           if (startsame(str[i], "positive"))
               con[ch_ind].positive = TRUE;
         else
           if (startsame(str[i], "negative"))
               con[ch_ind].negative = TRUE;
         else
           if (startsame(str[i], "linked"))
               con[ch_ind].link = ++nlinks;
         else {
           sprintf(err_string,
             "unknown type for constant %c, %s\n", ch, str[i]);
           if (put_error(err_string) == BREAK) {
                 return(BREAK);
           }
         }
```

```
    }
   }
   if (nlinks) {
    constraints = (struct constraint   *)
      emalloc ((unsigned int)(sizeof (struct constraint) * (nlinks +
1)));
    for (i = 0; i < nlinks; i++) {
        constraints[i].num = 0;
        constraints[i].max = MAXCONSTRAINTS;
        constraints[i].cvec = ((float **)
            emalloc((unsigned int)(sizeof(float *) *
MAXCONSTRAINTS)));
        constraints[i].ivec = ((float **)
            emalloc((unsigned int)(sizeof(float *) *
MAXCONSTRAINTS)));
        for (j = 0; j < nunits; j++) {
          constraints[i].cvec[j] = NULL;
          constraints[i].ivec[j] = NULL;
        }
    }
   }
   else {
    constraints = NULL;
   }
   positive_constraints = ((float **)
          emalloc((unsigned int)(sizeof(float *) * MAXCONSTRAINTS)));
   for (i = 0; i < MAXCONSTRAINTS; i++)
    positive_constraints[i] = NULL;

   negative_constraints = ((float **)
          emalloc((unsigned int)(sizeof(float *) * MAXCONSTRAINTS)));
   for (i = 0; i < MAXCONSTRAINTS; i++)
    negative_constraints[i] = NULL;
   return(CONTINUE);
}

change_lrate() {
   struct Variable *varp;
   int     i,
           j;

   if ((varp = lookup_var("lrate")) != NULL) {
    change_variable("lrate",(int *) varp);
   }
   else {
    return(put_error("BIG PROBLEM: lrate is not defined"));
   }

   if (epsilon != NULL) {
    for (i = 0; i < nunits; i++) {
        for (j = 0; j < num_weights_to[i]; j++) {
          if (epsilon[i][j] != 0.0)
```

```
                epsilon[i][j] = lrate;
            }
         }
        }
        if (bepsilon != NULL) {
         for (i = 0; i < nunits; i++) {
            if (bepsilon[i] != 0.0)
              bepsilon[i] = lrate;
         }
        }
}

/* given a defined system, we will write the matrix and the biases
   out to a file.  The file format is one floating point number per
line,
   with the weight matrix in row major format followed by the biases.
*/

write_weights() {
    int     i,j,end;
    char   *str = NULL;
    char fname[BUFSIZ];
    char *star_ptr;
    char tstr[40];
    FILE * iop;

    if (weight == NULL) {
     return(put_error("cannot save undefined network"));
    }

nameagain:

    str = get_command("weight file name: ");
    if (str == NULL) return(CONTINUE);
    strcpy(fname,str);
    if ( (star_ptr = index(fname,'*')) != NULL) {
     strcpy(tstr,star_ptr+1);
     sprintf(star_ptr,"%d",epochno);
     strcat(fname,tstr);
    }
    if ((iop = fopen(fname, "r")) != NULL) {
     fclose(iop);
        get_command("file exists -- clobber? ");
     if (str == NULL || str[0] != 'y') {
        goto nameagain;
     }
    }
    if ((iop = fopen(fname, "w")) == NULL) {
     return(put_error("cannot open file for output"));
    }

    for (i = 0; i < nunits; i++) {
```

```
        for (j = 0; j < num_weights_to[i]; j++) {
      fprintf(iop, "%f\n", weight[i][j]);
        }
    }

    if (bias) {
      for (i = 0; i < nunits; i++) {
      fprintf(iop, "%f\n", bias[i]);
      }
    }

    if (sigma) {
      for (i = 0; i < nunits; i++) {
      fprintf(iop, "%f\n", sigma[i]);
      }
    }

    (void) fclose(iop);
    return(CONTINUE);
}


read_weights() {
    int     i,j,end;
    register float *wt, *shwt, *wtend, *shend;
    char   *str = NULL;
    FILE * iop;
    if(!System_Defined)
       if(!define_system())
       return(BREAK);

    if (weight == NULL) {
     return(put_error("cannot restore undefined network"));
    }


    if((str = get_command("File name for stored weights: ")) == NULL)
     return(CONTINUE);

    if ((iop = fopen(str, "r")) == NULL) {
     sprintf(err_string,"Cannot open weight file %s.",str);
     return(put_error(err_string));
    }

    for (i = 0; i < nunits; i++) {
     if(num_weights_to[i] == 0) continue;
     for (j = 0; j < num_weights_to[i]; j++) {
       if (fscanf(iop, "%f", &weight[i][j]) == 0) {
       fclose(iop);
         return(put_error("weight file is not correct for this
network"));
       }
```

```
      }
    }

    end = nunits;

    if (bias != NULL) {
      for (i = 0; i < end; i++) {
     if (fscanf(iop, "%f", &bias[i]) == 0) {
         fclose(iop);
         return(put_error("weight file is not correct for this
network"));
      }
       }
    }

    if (sigma != NULL) {
      for (i = 0; i < end; i++) {
     if (fscanf(iop, "%f", &sigma[i]) == 0) {
         fclose(iop);
         return(put_error("weight file is not correct for this
network"));
      }
       }
    }

    (void) fclose(iop);
    update_display();
    return(CONTINUE);
}

/* realloc positive_constraints, negative_constraints, and link
constraints
   this is called whenever the allocated constraint lists run out of
   space for additional constraints  14-May-87 MAF / 15-May-87 JLM */

enlarge_constraints(con_index) int con_index; {
    if (con_index = ENLARGE_POS) {
     maxpos += 100;
     positive_constraints = ((float **) erealloc
       ((char *) positive_constraints,
           (unsigned int) ((maxpos - 100) * sizeof(float *)),
           (unsigned int) (maxpos * sizeof(float *))));
    }
    else if (con_index = ENLARGE_NEG) {
     maxneg += 100;
     negative_constraints = ((float **) erealloc
       ((char *) negative_constraints,
          (unsigned int) ((maxneg -100) * sizeof (float *)),
          (unsigned int) (maxneg * sizeof(float *))));
    }
    else {
     constraints[con_index].max += 100;
```

```
        constraints[con_index].cvec = ((float **) erealloc
       ((char *)constraints[con_index].cvec,
        (unsigned int)
          ((constraints[con_index].max - 100) * sizeof(float *)),
        (unsigned int)
          (constraints[con_index].max * sizeof(float *))));
        constraints[con_index].ivec = ((float **) erealloc
       ((char *)constraints[con_index].ivec,
        (unsigned int)
          ((constraints[con_index].max - 100) * sizeof(float *)),
        (unsigned int)
          (constraints[con_index].max * sizeof(float *))));
    }
}
```

Appendix 6

Modified SDMO Source Code for Hybrid Epoch-Pattern Method

```
MAINMOD.C
/*this file  is part of "neuron" It contains several general routines
needed in the  shared memory version of the simulator*/

#include <stdio.h>
#include "maindefmod.h"

extern double *lumped;
extern int lumpid;
extern int *misc;
extern int miscid;

calc_page(size)
int size;
{
int inc, pg, endof, xtra, nuend;
inc = 1;
pg = 4096;
endof = sbrk(0);
while (pg <= endof) {
     pg = 4096 * inc;
     ++inc;
}
xtra = (pg - endof) + size;
nuend = sbrk(xtra);
return(pg);

} /* end of calc_page */


quit()
{
shmdt(lumped);
shmctl(lumpid, IPC_RMID, 0);
shmdt(misc);
shmctl(miscid, IPC_RMID, 0);
}/*  end of quit */

MAINDEFMOD.H
/*
    The neuron will process   ----
                              \                    exp
                              /      w ( x   - x  )
                              ----    i    i    o
*/
```

```
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>

#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/mpadvise.h>
#define forkflag misc[0]
#define parentflag misc[1]
#define readforkflag misc[2]
#define procnetflag misc[3]
#define checkerrflag misc[4]
#define master misc[6]
#define slave misc[7]
#define calcdeldoneflag misc[8]
#define readnetflag misc[9]

#define patterr2 lumped[0]

#define toterr lumped[1]

extern int errno;
extern char *sys_errlist[];
static int shmemtemp;
double patterr;

typedef struct
       neuron_type{
       int     no_of_inputs;
      double  *input_list; /* can be impreoved..for time being this
                                        is included...*/
      double  *weight_list;  /* the actual weights of neurons*/
      double  *delwgt;      /* delta weight changes of neurons b.b.*/
      double  *del;  /* common to both processors */
      double  *del1;  /* proc 1 delta*/
      double  *del2;  /* proc 2 delta in sh mem  */
      double  x_zero;
      double  exp;
      double  threshold;
      double  (* transfer_function)();
      /* any function can be placed here..cool */
      double output;
      double error; /*added BB */
      double delta; /*added BB*/
      double *bias; /*added BB*/
      double delbias; /*added BB*/
      double delta1; /*added BB*/
      double *delta2; /*added BB*/
      double bia; /*added BB*/
```

```
                    }NEURON;
typedef NEURON *NEURON_PTR;

typedef struct
     layer_type{
      int     no_of_neurons;  /* on this the layer  */

      NEURON_PTR *this_neuron; /*pointer to neuron  pointers on layer
*/
                    } LAYER;
typedef LAYER *LAYER_PTR;


typedef struct
     network_type{
     double  *net_input_list;    /* to have structure clear */
     double  *net_output_list;
     int     no_of_layers;   /* on this network  */
     LAYER_PTR  *this_layer; /* pointer to layer  pointers
                                          on network */
     double  *net_input_list1;    /* to have structure clear */
     double  *net_output_list1;
                 } NETWORK;
typedef NETWORK *NETWORK_PTR;

NETWORK   current_network;   /* global holding current network */


typedef struct
          set_up{
              int    no_of_inputs;
              int    no_of_layers;
              int    *neurons_per_layer;
                } SET_UP_TYPE;

SET_UP_TYPE    network_set_up;

/*  Now define the global variables, they are not pointers because  */
/* when the program gets bigger compiler goes crazy and cannot store
*/
/*  string..*/

char   input_file_name[80], output_file_name[80],
       network_file_name[80], this_buffer[80];

int    global_cursor_y,  global_cursor_x;


MAINMOD.C
/*-------------------------------------------------------------------*/
/*   Module Name: Main.c MODIFIED FOR HYBRID mainmod.c               */
/*  ==============                                                   */
```

```
/*  This is the main body of the program.                              */
/*   Becker Co., Inc 1989 (C)                                          */
/*--------------------------------------------------------------------*/

#include <stdio.h>
#include  "maindefmod.h"

main(argc, argv)
int argc;
char **argv;

{ /* begin of MAIN */

   run_plain(argc,argv);
   exit(0);

}  /* end of MAIN */
```

BPTRNMOD.C

```
/*--------------------------------------------------------------------*/
/* Module Name: bptrnmod.c MODIFIED FOR HYBRID APPROACH               */
/* ===========================                                        */
/*      Becker Co., Inc. 1989 (C)                                     */
/*--------------------------------------------------------------------*/

/*
   NOTE:  The field output of a neuron cell will be used to store the
error
   fedback..just temporarily for next interation..
*/

#include  <curses.h>
#include  "maindefmod.h"

typedef struct  func_pass {
                            double   value;
                            double   thr;
                            double   (* this_function)();
                          }  FUNC_ARG_TYPE;

FILE  *fp, *fopen();
extern int *misc;
extern double *lumped;
int function, pid;

adjust_bp_error(this_network)
NETWORK  *this_network;

/* from the last layer will start backpropagating errors and adjusting

waits as it goes toward the first layer. Assumes that the error is
```

```
already backpropagated and stored on <.output> field of each neuron on

layer ...*/

{   /* begin of ADJUST_BP_ERROR  */
   int    i,  looper, neuron_loop;
  for ( i = this_network->no_of_layers - 1 ;
        i >= 0 ; --i)  /*don't adjust first layer on loop */
   {

  for ( neuron_loop = 0;
        neuron_loop < this_network->this_layer[i]->no_of_neurons;
        ++neuron_loop)
    {


     for (looper = 0;
          looper < this_network->this_layer[i]-
>this_neuron[neuron_loop]->no_of_inputs; ++looper)
     {

         this_network->this_layer[i]->this_neuron[neuron_loop]-
>delwgt[looper] =
 (0.5 * this_network->this_layer[i]->this_neuron[neuron_loop]-
>del[looper]) +     (0.9 * this_network->this_layer[i]-
>this_neuron[neuron_loop]->delwgt[looper]);


 this_network->this_layer[i]->this_neuron[neuron_loop]-
>weight_list[looper] +=
   this_network->this_layer[i]->this_neuron[neuron_loop]-
>delwgt[looper];

this_network->this_layer[i]->this_neuron[neuron_loop]->del[looper] =
0.0;

        } /* end of looper for */

  this_network->this_layer[i]->this_neuron[neuron_loop]->delbias =
  0.5 * this_network->this_layer[i]->this_neuron[neuron_loop]->bia +
                              0                .                9    *
this_network->this_layer[i]->this_neuron[neuron_loop]->delbias;

  this_network->this_layer[i]->this_neuron[neuron_loop]->bias[0] +=
  this_network->this_layer[i]->this_neuron[neuron_loop]->delbias;
  this_network->this_layer[i]->this_neuron[neuron_loop]->bia  = 0.0;


   } /* end of neuron_loop for */
 } /* end of i layer for loop */
}  /* end of ADJUST_BP_ERROR  LAYER  */
```

```
mod(this_network, this_file)
NETWORK *this_network;
FILE *this_file;
{
int i, neuron_loop, looper, dummy, no_of_patterns;
static int incr;

if(!readforkflag){
  readforkflag = 1;

 errno = 0;
  pid = fork();
  if(errno >0)
  fprintf(stderr, "Trouble with fork, errno = %3d, which means %s\n",
errno, sys_errlist[errno]);
  errno  = 0;
}/*  end of not fork */


if(pid == 0){
errno = 0;
function = mpadvise(MPA_CPU_SET, 4);
if(errno >0)
fprintf(stderr, "Trouble with fork, errno = %3d, which means %s\n",
errno, sys_errlist[errno]);
errno = 0;
rewind(this_file);
fscanf(this_file, "%d\n", &no_of_patterns);
fscanf(this_file, "%d\n", &dummy);
fscanf(this_file, "%d\n", &dummy);

modagain:
if(incr == no_of_patterns){
rewind(this_file);
fscanf(this_file, "%d\n", &no_of_patterns);
fscanf(this_file, "%d\n", &dummy);
fscanf(this_file, "%d\n", &dummy);
incr = 0;
} /* end of if incr */
for(looper = 0; looper <this_network->this_layer[0]->this_neuron[0]-
>no_of_inputs; ++looper)
{
fscanf(this_file, "%lf\n", &this_network->net_input_list[looper]);
}

for(looper = 0; looper <this_network->this_layer[this_network-
>no_of_layers -1] ->no_of_neurons; ++ looper)
{
fscanf(this_file, "%lf\n", &this_network->net_output_list[looper]);
}

readnetflag = 1;
```

```
incr += 1;
for(looper = 0; looper <this_network->this_layer[0]->this_neuron[0]-
>no_of_inputs; ++looper)
{
fscanf(this_file, "%lf\n", &this_network->net_input_list1[looper]);
}

for(looper = 0; looper <this_network->this_layer[this_network-
>no_of_layers -1] ->no_of_neurons; ++ looper)
{
fscanf(this_file, "%lf\n", &this_network->net_output_list1[looper]);
}


process_network_b(this_network);
while(!procnetflag);
procnetflag = 0;

check_error_output(this_network);


  for ( i=0; i <= this_network->no_of_layers - 1 ; i++){
  for ( neuron_loop = 0;
        neuron_loop < this_network->this_layer[i]->no_of_neurons;
        ++neuron_loop)
    {

     for (looper = 0;
          looper < this_network->this_layer[i]-
>this_neuron[neuron_loop]->no_of_inputs;
          ++looper)
     {

 this_network->this_layer[i]->this_neuron[neuron_loop]->del2[looper]
=
 this_network->this_layer[i]->this_neuron[neuron_loop]->delta *
this_network->this_layer[i]->this_neuron[neuron_loop]-
>input_list[looper];

this_network->this_layer[i]->this_neuron[neuron_loop]->delta2[0] =
 this_network->this_layer[i]->this_neuron[neuron_loop]->delta;

}/*  end of i */
} /* end of neurloop */
}/*  end of looper */
incr += 1;
 calcdeldoneflag = 1;
while(!parentflag);
parentflag = 0;
goto modagain;

} /*end of if */
```

```
if(pid !=0) return;
} /* end of MOD*/


calcdel(this_network)
NETWORK *this_network;
{ /* begin calcdel*/
int i,neuron_loop,looper;

for ( i=0; i <= this_network->no_of_layers - 1 ; i++){
  for ( neuron_loop = 0; neuron_loop < this_network->this_layer[i]-
>no_of_neurons; ++neuron_loop)
    {

     for (looper = 0;
     looper < this_network->this_layer[i]->this_neuron[neuron_loop]-
>no_of_inputs; ++looper)
     {

 this_network->this_layer[i]->this_neuron[neuron_loop]->del1[looper]
=
 this_network->this_layer[i]->this_neuron[neuron_loop]->delta *
this_network->this_layer[i]->this_neuron[neuron_loop]-
>input_list[looper];


this_network->this_layer[i]->this_neuron[neuron_loop]->delta1 =
 this_network->this_layer[i]->this_neuron[neuron_loop]->delta;

   }
  }
}

} /* end of cacdel  */


calcbia(this_network)
NETWORK *this_network;
{ /*BEGIN of calcbia */
int i, neuron_loop, looper ;
  for ( i=0; i <= this_network->no_of_layers - 1 ; i++){
  for ( neuron_loop = 0; neuron_loop < this_network->
       this_layer[i]->no_of_neurons;++neuron_loop)
    {

     for (looper = 0;looper < this_network->
          this_layer[i]->this_neuron[neuron_loop]->no_of_inputs;
          ++looper)
     {

 this_network->this_layer[i]->this_neuron[neuron_loop]->del[looper]=
 this_network->this_layer[i]->this_neuron[neuron_loop]->del1[looper]
```

```
+
this_network->this_layer[i]->this_neuron[neuron_loop]->del2[looper];

this_network->this_layer[i]->this_neuron[neuron_loop]->bia =
 this_network->this_layer[i]->this_neuron[neuron_loop]->delta1 +
 this_network->this_layer[i]->this_neuron[neuron_loop]->delta2[0];

}
}
}
}/* end of calcbia*/


check_error_output(this_network)
NETWORK   *this_network;
/* Compare the output of net on training if it's on limit the stop
   and also update the output field on last layer of network.
   Returns 1 if output vector is on range to be considered close to
...*/

{   /* begin of  CHECK_ERROR_OUTPUT   */


   int  looper, neuron_loop;
   double temp_err, temp, sigmoid(), derivative();


   int inner, outer;
   int lay, i, j, inputs;

   FUNC_ARG_TYPE  dummy;
for (outer = 0; outer < this_network->no_of_layers; ++outer)
    for( inner=0; inner< this_network->this_layer[outer]-
>no_of_neurons;++inner)
  this_network->this_layer[outer]->this_neuron[inner]->error = 0.0;

   patterr = patterr2 = 0.0;

for(looper = 0;
    looper < this_network->this_layer[this_network->no_of_layers - 1]
             ->no_of_neurons; ++ looper)
{

if(pid != 0){
temp_err =  this_network->this_layer[this_network->no_of_layers - 1]-
>this_neuron[looper]
   ->error = this_network->net_output_list[looper] -
         this_network->this_layer[this_network->no_of_layers - 1]
               ->this_neuron[looper]->output;
patterr += temp_err * temp_err;

}
```

```
else  {

temp_err =  this_network->this_layer[this_network->no_of_layers - 1]-
>this_neuron[looper]
   ->error = this_network->net_output_list1[looper] -
         this_network->this_layer[this_network->no_of_layers - 1]
               ->this_neuron[looper]->output;

patterr2 += temp_err*temp_err;

}
}
for(lay = this_network->no_of_layers -1; lay >=0; lay --)
   {
   for(i=0;i<this_network->this_layer[lay]->no_of_neurons; i++)
   {
temp = this_network->this_layer[lay]->this_neuron[i]->output;
   this_network->this_layer[lay]->this_neuron[i]->delta =
   this_network->this_layer[lay]->this_neuron[i]->error * temp * (1.0
-
temp);

   for(j=0;j<this_network->this_layer[lay]->this_neuron[i]-
>no_of_inputs; j++)
   {
    if (lay ==0)
      inputs = 1;
    else
    {
    this_network->this_layer[lay-1]->this_neuron[j]->error +=
    this_network->this_layer[lay]->this_neuron[i]->delta *
    this_network->this_layer[lay]->this_neuron[i]->weight_list[j];


} /* end else */
    } /* end j for loop */
} /* end i for loop */
} /* end lay for loop */

}   /* end of CHECK_ERROR_OUTPUT   */


back_prop_train(this_network, this_alpha)
NETWORK   *this_network;
double    this_alpha;
/* Assumes that data is already placed on the net_output/input_list
   vector on network... This function is for a plain terminal*/
{   /* begin of BACK_PROP_TRAIN   */

  extern   char *read_string(), *check_file();
  FILE     *temp, *fopen();
  int   loo, cursor_y, cursor_x, looper, dummy, no_of_patterns,
```

```
no_of_passes=0;
  int      fine, no_trials, t, done, i, step = 0, trials = 0;
  register double temp_err;

  int no_of_process = 2, no_of_pats;

  char     *fname, buff[80], this_choice;
  WINDOW   *back_screen, *error_screen , *create_window();

  parentflag = 0;
  forkflag = 0;

  initscr();
  back_screen = create_window(20,70, 4,4);
  error_screen = create_window(2,70, 22,4);


  insert_string( back_screen,
    "Will the patterns be entered from keyboard/file (k/f)?",1,2,0);
  wrefresh(back_screen);
  getyx( back_screen, cursor_y, cursor_x);
  this_choice = read_key_stroke(back_screen);

  if( this_choice != 'f' && this_choice != 'k')
  {
     error_message(back_screen, error_screen ,"Wrong input..",
                   cursor_y, cursor_x);
     this_choice = read_key_stroke(back_screen);
     werase(error_screen);
  }
  wrefresh(error_screen);
  insert_string(back_screen,"What is the pattern file name?",4,2,0);
  wrefresh(back_screen);
  getyx(back_screen,cursor_y, cursor_x);
  fname = read_string(back_screen);
  fname = check_file(fname, "r", back_screen, error_screen,

                                  cursor_y, cursor_x);
  strcpy(buff ,fname);

  check_pattern_file( buff, this_network);

  temp = fopen(buff, "r");

insert_string(back_screen, "For how many trials do you want to
train?",
6,2,0);
wrefresh(back_screen);
getyx(back_screen, cursor_y, cursor_x);
no_trials   =   get_integer(back_screen,   error_screen,   cursor_y,
cursor_x);
wrefresh(back_screen);
```

```
getyx(back_screen, cursor_y, cursor_x);
 step = answer_yes_no(back_screen, error_screen,
  "Do you want to see all the steps (y/n)?", cursor_y+1, 1);
wrefresh(back_screen);
trials = answer_yes_no(back_screen, error_screen,
                  "Do you want to see all the trials (y/n)?",
cursor_y+2, 1);
werase(back_screen);
rewind(temp);
fscanf(temp, "%d\n", &no_of_patterns);
fscanf(temp, "%d\n", &dummy);
fscanf(temp, "%d\n", &dummy);
     print_network_stat(this_network, back_screen);

no_of_pats = no_of_patterns;
if(no_of_patterns % no_of_process == 0)
no_of_patterns = no_of_patterns/no_of_process;
else
no_of_patterns = no_of_patterns/no_of_process +1;

do_it_again:

for(t = 0; t < no_trials; ++t)
{
     no_of_passes += 1; /* incrememt the counter of the trainings */

if (trials){
 wrefresh(back_screen);

     wmove(back_screen, 5,4);

     wprintw(back_screen,
            "Network has gone through  %d  trials \n", no_of_passes);

}
 toterr = 0.0;
   for( looper =0;
        looper <  no_of_patterns;
        ++looper)
   {
if (trials){

      wmove(back_screen, 6,4);
      wprintw(back_screen,
   "Training file has  %d   patterns, current pattern is  %d\n",
                  no_of_pats ,  looper + 1);
}

  if (!readforkflag)
mod(this_network, temp);
while(!readnetflag);
readnetflag = 0;
```

```
process_network_b(this_network);
procnetflag = 1;

check_error_output(this_network);
if(trials){

       for(loo = 0;
           loo  <this_network->this_layer[this_network->no_of_layers
-1]
                   ->no_of_neurons;
           ++loo)
       {
         wmove( back_screen, 9 + loo , 2);
         wprintw(back_screen, " Item[ %d ] = %f",
          loo,   this_network->this_layer[this_network->no_of_layers
-1]
                                   ->this_neuron[loo]->output);
    wmove( back_screen, 9 + loo , 25);
         wprintw(back_screen, "Desired Item[ %d ] = %f\n",
                loo,   this_network->net_output_list[loo]);
        }
}/*  end of if trials*/
calcdel(this_network);
while(!calcdeldoneflag);
calcdeldoneflag = 0;
toterr += patterr2 + patterr;
calcbia(this_network);
adjust_bp_error(this_network);
parentflag = 1;
if(trials)
{
      wmove(back_screen, 7,4);
      wprintw(back_screen, "patterr = %f", patterr);
      wmove(back_screen, 7,24);
      wprintw(back_screen, "Toterr = %f\n", toterr);

}/*  end of trial if*/

if(step){
       wmove(back_screen, 16,5);
       wprintw(back_screen,  " %d BP iterations \n", no_of_passes);
       wprintw(back_screen,   "Press   Any   key   to   continue",
no_of_passes);
       wrefresh(back_screen);
       getch();
}
  } /* end of no of patterns FOR  */


     if (toterr < 0.04){
     fine = 1;
        break;
```

```
        }

} /* end of no_trials FOR */
kill(pid, SIGKILL);
readforkflag = 0;
if(!trials){
 wrefresh(back_screen);

     wmove(back_screen, 5,4);

     wprintw(back_screen,
            "Network has gone through  %d  trials \n", no_of_passes);

      wmove(back_screen, 6,4);
      wprintw(back_screen,
      "Training file has  %d   patterns, current pattern is  %d\n",
                  no_of_pats ,  looper);
       for(loo = 0;
           loo  <this_network->this_layer[this_network->no_of_layers
-1]
                   ->no_of_neurons;
           ++loo)
       {
         wmove( back_screen, 9 + loo , 2);
         wprintw(back_screen, " Item[ %d ] = %f",
          loo,   this_network->this_layer[this_network->no_of_layers
-1]
                                    ->this_neuron[loo]->output);
    wmove( back_screen, 9 + loo , 25);
         wprintw(back_screen, "Desired Item[ %d ] = %f\n",
                loo,   this_network->net_output_list[loo]);

        }

    wmove(back_screen, 7,4);
      wprintw(back_screen, "patterr = %f", patterr);
      wmove(back_screen, 7,24);
      wprintw(back_screen, "Toterr = %f\n", toterr);
} /* end of not trials IF */

if(fine == 1){
wmove(back_screen, 16,0);
 wprintw(back_screen, "FINISHED, solution found after %d training
sequences\n", no_of_passes);
 wrefresh(back_screen);
}

    wmove(back_screen, 18,5);
    getyx(back_screen,cursor_y, cursor_x);

if(answer_yes_no(back_screen, error_screen,
                "Do you want stats (y/n)?", cursor_y, cursor_x) ==
```

```
1){

      system ("clear");
      rstats(stdout);
      wmove(back_screen, 17,0);
      getyx(back_screen, cursor_y, cursor_x);
}
   if(  answer_yes_no(back_screen, error_screen,
 " Would you like to train again (y/n)?", cursor_y, cursor_x) == 1){

       system("clear");
     initscr();
        if (done==1)
           no_of_passes = 0;
                forkflag = 0;
     goto do_it_again;
   }
 endwin();

}   /* end of BACK_PROP_TRAIN  */


NEURTMOD.C
/*-------------------------------------------------------------------*/
/*Module Name:  neurtulmod.c MODIFIES FOR HYBRID APPROACH           */

/*============================                                      */
/*   Becker Co., Inc. 1989 (C)                                      */
/*-------------------------------------------------------------------*/


#include <stdio.h>
#include <malloc.h>
#include "maindefmod.h"

#define TF_VALUE_L  0.0
#define TF_VALUE_H  1.0

extern int pid;

int lumpid, lumppage;
unsigned lumpsize;
double *lumped = NULL;

int miscid, miscpage;
unsigned miscsize;
int *misc = NULL;
int ioid;

typedef struct  func_pass {
                          double   value;
                          double   thr;
```

```
                        double   (* this_function)();
                      }  FUNC_ARG_TYPE;

FILE *fopen(), *fp;


/*
  This set of routines create (dynamically) a neural
  network of any size the only limitation is the machine
  that is running it.

  It will be developed as device independent as possible so that
  to make it run on any machine all it would be needed will be
  the drivers for that particular machine (for the graphics ouput)

     by Becker Co, Inc (1989) (C)
*/

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 1
-----------------------------------*/

NEURON create_neuron( no_of_input)
int no_of_input;
{
   NEURON  temp_neuron;
   int     looper;
   double  sharp(), sigmoid();
   temp_neuron.input_list = (double *)calloc(no_of_input,
                                             sizeof(double));
/*   temp_neuron.weight_list = (double *)calloc(no_of_input,
                                             sizeof(double));
*/
     temp_neuron.delwgt = (double *)calloc(no_of_input,
                                             sizeof(double));
     temp_neuron.del = (double *)calloc(no_of_input,
                                             sizeof(double));
     temp_neuron.del1 = (double *)calloc(no_of_input,
                                             sizeof(double));

temp_neuron.weight_list = &lumped[shmemtemp];
shmemtemp += no_of_input +2;
temp_neuron.del2 = &lumped[shmemtemp];
shmemtemp += no_of_input +2;
temp_neuron.bias = &lumped[shmemtemp];
shmemtemp += 2;
temp_neuron.delta2 = &lumped[shmemtemp];
shmemtemp += 2;

   /* Now that we allocate space for the weights..initialize randomly
*/
   for ( looper =0 ; looper < no_of_input ; ++looper)
{     temp_neuron.weight_list[looper] =( rand()* 6.1035E-5) - 1.0;
```

```
      temp_neuron.delwgt[looper] = 0.0;
      temp_neuron.del[looper] = 0.0;
      temp_neuron.del1[looper] = 0.0;
      temp_neuron.del2[looper] = 0.0;
}

   /* compiler doen't like undefined values..so intialize everything
*/

   temp_neuron.no_of_inputs = no_of_input;

   temp_neuron.x_zero = 0.0;
   temp_neuron.exp = 1.0;
   temp_neuron.threshold = 0.0;
   temp_neuron.transfer_function = sigmoid ; /* default sharp TF */

   temp_neuron.output    = 0.0;
   temp_neuron.error = 0.0;
   temp_neuron.delta = 0.0;
   temp_neuron.bias[0] = ( rand()*6.1035E-5) - 1.0;
   temp_neuron.delbias = 0.0;
   temp_neuron.delta1 = 0.0;
   temp_neuron.delta2[0] = 0.0;
   temp_neuron.bia = 0.0;


   return(temp_neuron);
}
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 2
-----------------------------------*/

LAYER create_layer( layer_no, no_of_neurons, prev_layer_no_of_neurons)
int layer_no;
int no_of_neurons;
int prev_layer_no_of_neurons;

{  /* begin of CREATE_LAYER */
   NEURON      create_neuron();

   LAYER       temp_layer;
   int         i;
   NEURON       *dummy_neuron; /* memory space MUST be allocated to
hold
                                  the actual neuron space  */

    /*  Create dynamically the pointers to neurons*/
    temp_layer.this_neuron = (NEURON_PTR *)calloc(no_of_neurons,

sizeof(NEURON_PTR));

  dummy_neuron  = (NEURON *)calloc(no_of_neurons, sizeof(NEURON) );
```

```
    temp_layer.no_of_neurons = no_of_neurons;

    /* Now create the actual neurons  */
    for (i=0; i < no_of_neurons ; ++i)
    {
       dummy_neuron[i] = create_neuron(prev_layer_no_of_neurons);
       temp_layer.this_neuron[i] = &dummy_neuron[i];
    }

}  /* end of CREATE_LAYER */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 3
-----------------------------------*/

NETWORK create_network(net_set_up)
SET_UP_TYPE net_set_up;

{  /* begin of CREATE_NETWORK */

     NETWORK network_temp;
     int     counter;
         LAYER *dummy_layer;

int i, totneuron, connects;
totneuron = 0;
connects = 0;

/*the following calcs the size of the network then creates enough
shared memory space to handle the parallel procesing */

for(i = 0; i < net_set_up.no_of_layers; i++)
{
     totneuron += net_set_up.neurons_per_layer[i];
     if (i == 0)
        connects = net_set_up.no_of_inputs *
net_set_up.neurons_per_layer[i];
     else
        connects += net_set_up.neurons_per_layer[i-1] *
                    net_set_up.neurons_per_layer[i];
} /* end of for */

connects= (net_set_up.neurons_per_layer[0] *3 ) +
(net_set_up.neurons_per_layer[(net_set_up.no_of_layers - 1)]*3)  +
(connects * 4) + (totneuron * 5) + 4;

errno = 0;

lumpsize = (unsigned)(sizeof(double) *  connects );
fprintf(stderr, "connects = %d, lumpsize = %d\n", connects, lumpsize);
lumpid = shmget(0, lumpsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "lumped errno is %3d, which means %s\n", errno,
```

```
sys_errlist[errno]);
errno = 0;
lumppage = calc_page(lumpsize);
lumped = (double *) shmat(lumpid, lumppage, 0);
if (errno > 0)
fprintf(stderr, "In lumped  %s\n", sys_errlist[errno]);
for( i = 0; i< connects ; i++)
     lumped[i] = 0.0;

/*create enough space for ten flags to be used in bp_train.c-- see
define section */
miscsize = (unsigned)(sizeof(int) *  10 );
miscid = shmget(0, miscsize, 0666|IPC_CREAT);
if (errno >0)
fprintf(stderr, "misc errno is %3d, which means %s\n", errno,
sys_errlist[errno]);
errno = 0;
miscpage = calc_page(miscsize);
misc = (int *) shmat(miscid, miscpage, 0);

if (errno > 0)
fprintf(stderr, "In misc  %s\n", sys_errlist[errno]);
for( i = 0; i< 10 ; i++)
     misc[i] = 0;
shmemtemp = 2;

     /* Create fist the pointers to the layers of the network */
     network_temp.this_layer = (LAYER_PTR *)
      calloc(net_set_up.no_of_layers, sizeof(LAYER_PTR));
    dummy_layer = (LAYER *)calloc(net_set_up.no_of_layers,
sizeof(LAYER) );

     network_temp.no_of_layers = net_set_up.no_of_layers;

     for ( counter=0; counter < net_set_up.no_of_layers ; ++counter)
        {
           if ( counter == 0)   /* first layer */
        {
              dummy_layer[counter] = create_layer(counter,
                                net_set_up.neurons_per_layer[counter],
                                net_set_up.no_of_inputs);
                    network_temp.this_layer[counter]    =
&dummy_layer[counter];

        }
           else{
               dummy_layer[counter] = create_layer( counter,
                            net_set_up.neurons_per_layer[counter],
                           net_set_up.neurons_per_layer[counter-1]);
               network_temp.this_layer[counter] =
                &dummy_layer[counter];
             }
```

```
        }

        /* Now create the input and output array (dynamically) as
requested by user...to be used to train or output nicely*/

      network_temp.net_input_list = &lumped[shmemtemp];
      shmemtemp+= net_set_up.neurons_per_layer[0] + 10;

      network_temp.net_output_list = &lumped[shmemtemp];
shmemtemp+=net_set_up.neurons_per_layer[(net_set_up.no_of_layers-1)]
            + 10;

      network_temp.net_input_list1 = &lumped[shmemtemp];
      shmemtemp+= net_set_up.neurons_per_layer[0] + 10;

      network_temp.net_output_list1 = &lumped[shmemtemp];
 shmemtemp+=   net_set_up.neurons_per_layer[(net_set_up.no_of_layers
-1)]
             + 10;
         return ( network_temp);

}  /* begin of CREATE_NETWORK */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 4
----------------------------------*/

/*      Now a set of routines to read  and write the current
*/
/*      network ..so work is not lost
*/
/*-------------------------------------------------------------------
-*/
save_neuron(this_neuron, this_file)
NEURON    *this_neuron;
FILE      *this_file;

/* assumes that file was already open in calling function..just saves
   the neuron in order..i.e all fileds  */
{   /* begin of SAVE_NEURON   */
   int temp_index;

   fprintf(this_file,"%d\n", this_neuron->no_of_inputs);
   fprintf(this_file,"%f\n", this_neuron->bias[0]);

/* finally save the weights  */

   for ( temp_index =0; temp_index < this_neuron->no_of_inputs;
         ++temp_index)
   {
              f p r i n t f ( t h i s _ f i l e , " % f \ n " ,
this_neuron->weight_list[temp_index]);
```

```
   }

}   /* end of SAVE_NEURON     */
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 5
-----------------------------------*/

save_network(this_network, file_name)
NETWORK   *this_network;
char      *file_name;

{  /* begin of SAVE_NETWORK  */
   int   layer_index, neuron_index;
   FILE  *this_file;

   if ( (this_file = fopen(file_name , "w") ) == NULL)
   {
     printf("Sorry cannot write on this file...\n");
     exit(1);
   }

  /* start by saving the network set up..*/
   fprintf(this_file, "%d\n" , this_network->this_layer[0]-
>this_neuron[0]->no_of_inputs);
   fprintf(this_file, "%d\n" , this_network->no_of_layers);

   for ( layer_index =0;
         layer_index < this_network->no_of_layers;
         ++layer_index)
   {
 fprintf( this_file, "%d\n"/* = neurons on layer-------- [ %d ] \n"*/,
          this_network->this_layer[layer_index]->no_of_neurons/*,
          layer_index*/);
   }


   /* Now save the network ..every neuron from 0 -> no  */
   for ( layer_index =0;
         layer_index < this_network->no_of_layers;
         ++layer_index)
  {
      for ( neuron_index =0;
            neuron_index < this_network->this_layer[layer_index]-
>no_of_neurons;
            ++neuron_index)
      {
          save_neuron(this_network->this_layer[layer_index]->
                        this_neuron[neuron_index],this_file );
      }

  }

}  /* end of SAVE_NETWORK  */
```

```
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 6
-----------------------------------*/


read_neuron(this_neuron, this_file)
NEURON   *this_neuron;
FILE     *this_file;

/* assumes that file was already open in calling function..just saves
   the neuron in order..i.e all fields  */
{   /* begin of READ_NEURON    */
   int temp_index;

   fscanf(this_file,"%d\n", &this_neuron->no_of_inputs);
   fscanf(this_file,"%lf\n", &this_neuron->bias[0]);
   /* finally read the weights  */

   for ( temp_index =0; temp_index < this_neuron->no_of_inputs;
         ++temp_index)
   {
       fscanf(this_file,"%lf\n", &this_neuron-
>weight_list[temp_index]);
   }

}   /* end of READ_NEURON      */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 7
-----------------------------------*/

read_network(this_network, file_name)
NETWORK   *this_network;
char  *file_name;

/* Note that to be on the safe side when calling this function it's
   better... BELIEVE ME..to send in a buffer[80] instead of just the
   pointer for 'file_name' otherwise it's not reliable  */


{  /* begin of READ_NETWORK  */
   int          layer_index, neuron_index;
   FILE         *fopen(), *this_file;
   SET_UP_TYPE  temp_set_up;
   NETWORK      create_network();

  if ( (this_file = fopen(file_name , "r")) == NULL)
  {
     printf("Sorry cannot read on this file...\n");
     exit(1);
   }

   rewind(this_file);
```

```
  /* start by reading the network set up..*/
   fscanf(this_file, "%d\n", &temp_set_up.no_of_inputs);
   fscanf(this_file, "%d\n", &temp_set_up.no_of_layers);

   temp_set_up.neurons_per_layer = (int *)calloc(
temp_set_up.no_of_layers, sizeof(int)  );
   for ( layer_index =0;
         layer_index < temp_set_up.no_of_layers;
         ++layer_index)
   {
      fscanf( this_file, "%d\n ",
              &temp_set_up.neurons_per_layer[layer_index]);
/*fprintf(stderr, "neurons  %d\n",
temp_set_up.neurons_per_layer[layer_index]);
*/
}

   /* Once the set up is there..create the memory space to hold the
      network to be read...*/
      *this_network = create_network(temp_set_up);

   /* Now read the network ..every neuron from 0 -> no  */
   for ( layer_index =0;
         layer_index < this_network->no_of_layers;
         ++layer_index)
  {
      for ( neuron_index =0;
            neuron_index < this_network->this_layer[layer_index]-
>no_of_neurons;
            ++neuron_index)
      {
          read_neuron(this_network->this_layer[layer_index]->
                                        this_neuron[neuron_index],
                      this_file );
     }

  }

}  /* end of READ_NETWORK  */



check_pattern_file(file_name, for_this_network)
char     *file_name;
NETWORK  *for_this_network;
/*
   Will scan the file with the pattern(s) and check that all data is
   consistent..so when training starts it won't hang ...Assumes that
the file exits when called.. Outputs 0 on success, 1 on failure..
*/
{   /* begin of CHECK_PATTERN_FILE */
```

```
   FILE  *this_file, *fopen();
   int      no_of_patterns, no_of_inputs, no_of_outputs,
            looper, counter ;
double   temp_value;

   /* Check first if the file exists */
   if ( (this_file = fopen(file_name , "r")) == NULL)
   {
     printf("Sorry cannot read this file...\n");
     exit(1);
   }

   rewind(this_file);

   /* Now that the file is there check if patterns are correct,
       i.e. dimesions and format  */
   fscanf(this_file, "%d\n", &no_of_patterns);
   fscanf(this_file, "%d\n", &no_of_inputs);
   fscanf(this_file, "%d\n", &no_of_outputs);
   if ( (no_of_inputs != for_this_network->this_layer[0]-
>this_neuron[0] ->no_of_inputs )   ||
 (no_of_outputs != for_this_network->this_layer[for_this_network
 ->no_of_layers -1]->no_of_neurons) )
   {
     printf("Sorry patterns dimensions do not match network's...\n");
     exit(1);
   }

   /* Finally check that file contains patterns correctly
      doesn't check when incomplete data... */

   for (looper =0; looper < no_of_patterns; ++looper)
   {
        for ( counter =0;
              counter < no_of_inputs;
              ++ counter)
          if ( fscanf(this_file,"%lf\n", &temp_value) == EOF)
           {
              printf( " Sorry..incomplete data  on file...\n");
              exit(1);
           }

        for (counter=0; counter < no_of_outputs ; ++counter)

        {
           if ( fscanf(this_file,"%lf\n", &temp_value) == EOF)
           {
              printf( " Sorry..incomplete data  on file...\n");
              exit(1);
           }
        }
   }
```

```
   /* If everything went correct then the file has right information
      return succesful code..*/
   return(0);

}   /* end of CHECK_PATTERN_FILE */

read_network_io(this_network,  this_file)
NETWORK  *this_network;
FILE     *this_file;
/* Assumes that file already exists and that it's already open so
   it's just ready to read the input and ouput from current position
    on file...
*/
{  /*  begin of READ_NETWORK_IO  */
   int looper;
int dummy, no_of_patterns;
static int incr;
if(!readforkflag){
readforkflag = 1;

errno = 0;
ioid = fork();
if(errno>0)
fprintf(stderr, "trouble with fork, errno = %3d, which means %s\n",
errno, sys_errlist[errno]);
errno = 0;
}
if(ioid == 0){
fprintf(stderr, "ioid is %d\n", ioid);

rewind (this_file);
fscanf(this_file, "%d\n", &no_of_patterns);
fscanf(this_file, "%d\n", &dummy);
fscanf(this_file, "%d\n", &dummy);
readanother:
while (!master && !slave);
if (incr == no_of_patterns){
rewind (this_file);
fscanf(this_file, "%d\n", &no_of_patterns);
fscanf(this_file, "%d\n", &dummy);
fscanf(this_file, "%d\n", &dummy);
incr = 0;
} /* end of if */

   /* read the input vector and place  it on network */
   for (looper=0;

        looper < this_network->this_layer[0]->this_neuron[0]
                                    ->no_of_inputs  ;
        ++looper)
{
if(!master) {
```

```
      fscanf(this_file, "%lf\n", &this_network-
>net_input_list1[looper]);
}
else
      fscanf(this_file, "%lf\n", &this_network-
>net_input_list[looper]);
}
   /* read the output vector and place  it on network */
   for (looper=0;
 looper < this_network->this_layer[this_network->no_of_layers -1]
  ->no_of_neurons ; ++looper){
     if(!master) {
    fscanf(this_file, "%lf\n", &this_network-
>net_output_list1[looper]);

}
else
 fscanf(this_file, "%lf\n", &this_network->net_output_list[looper]);

}
master = slave = 0;
incr += 1;
readnetflag = 1;
goto readanother;
} /* end of fork loop */
if(ioid != 0) return;

}  /*  end of READ_NETWORK_IO  */

/*  Now a set of transfer functions that can be used as transfer
function*/

double sharp(this_x, this_threshold)
double   this_x;
double   this_threshold;

{   /* begin of SHARP */
  if (  this_x >= this_threshold)
    return(TF_VALUE_H);
  else
   return(TF_VALUE_L);
}  /* end of SHARP   */


double sigmoid( this_x, this_threshold)
double   this_x;
double   this_threshold;
{   /* begin of SIGMOID  */
   double exp();

  double limit = 12.0;
```

```
  /* just in case convert to double everything..*/
   if ( this_x - this_threshold > limit)
      return( 0.9999998);
   else
     if ( this_x - this_threshold < -limit)
      return( 0.0000012);


    else
   return(  ( 1.0/(1.0 +  exp((-1.0)* (this_x - this_threshold)))
)       );
}   /* end of SIGMOID   */


double  derivative(this_func)
FUNC_ARG_TYPE   this_func;
{  /* begin of DERIVATIVE  */
   double  temp, ddelta = 1.0E-10;

   temp =  ( this_func.this_function(this_func.value + ddelta,
this_func.thr) -
             this_func.this_function(this_func.value, this_func.thr)
           ) / ddelta;

   return(temp);

}  /* end of DERIVATIVE   */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 7
-----------------------------------*/
/*      Now a set of routine to process the network                 */
/*                                                                  */

send_neuron_output(this_value, on_this_network,  from_layer_index,
                   from_neuron_index, to_neuron_index)
double  this_value;
NETWORK *on_this_network;
int     from_layer_index;
int     from_neuron_index;
int     to_neuron_index;

/* Calling routine should check that the last layer doesn't send
output
  or this routine will do nothing...*/

{  /* begin of SEND_NEURON_OUTPUT   */

   if ( (from_layer_index +1) >= on_this_network->no_of_layers)
     {
        printf("  Sorry no next layer to send values to..");
        exit(1);
     };
```

```
   on_this_network->this_layer[from_layer_index + 1]

              ->this_neuron[to_neuron_index]-
>input_list[from_neuron_index]= this_value;
}  /* end of SEND_NEURON_OUTPUT  */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 8
----------------------------------*/

send_layer(on_this_network, this_layer_index)
NETWORK  *on_this_network;
int      this_layer_index;

 /* This function will send values from layer to next ..in (hidden)
layers,
    for last layer use other routine   */

{   /* begin of SEND_LAYER  */
    int  from_ctr, to_ctr, from_tmp, to_tmp;

   /* if ( (this_layer_index == 0) ||   */
      if   (this_layer_index >= (on_this_network->no_of_layers -1) )
    {
       printf(" Sorry cannot process this layer...");
       exit(1);
    }

  /* We waste a little memory (~4bytes) but speed is gained ...*/
  from_tmp = on_this_network->this_layer[this_layer_index]-
>no_of_neurons ;
  to_tmp   = on_this_network->this_layer[this_layer_index +1]-
>no_of_neurons ;

  for ( from_ctr =0; from_ctr < from_tmp; ++from_ctr)
    for ( to_ctr = 0; to_ctr < to_tmp; ++to_ctr)
   {
     send_neuron_output(
         on_this_network->this_layer[this_layer_index]-
>this_neuron[from_ctr]-> output, on_this_network, this_layer_index,
         from_ctr, to_ctr);

   }

}  /* end of SEND_LAYER  */
/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 9
----------------------------------*/

send_outer_layer(on_this_network, first_or_last)
NETWORK  *on_this_network;
int      first_or_last;

{   /* begin  of SEND_OUTER_LAYER  */
```

```
   int   counter, counter1, temp;

   switch( first_or_last)
  {
  case 0: temp = 0;
         for ( counter=0;

         counter < on_this_network->this_layer[0]->no_of_neurons ;
               ++counter)

           for ( counter1=0;
            counter1 < on_this_network->this_layer[0]->
            this_neuron[0]->no_of_inputs ; ++counter1) {
     if(pid !=0)
           on_this_network->this_layer[0]->this_neuron[counter]
              ->input_list[counter1]=
                     on_this_network->net_input_list[counter1];
     else
           on_this_network->this_layer[0]->this_neuron[counter]
              ->input_list[counter1]=
                     on_this_network->net_input_list1[counter1];
           };

           break;

  default:    /* actually the last layer */
           temp = on_this_network->no_of_layers - 1;
           for ( counter =0;
           counter < on_this_network->this_layer[temp]->no_of_neurons
;
                 ++counter)
           {
              on_this_network->net_output_list[counter] =
                 on_this_network->this_layer[temp]-
>this_neuron[counter] ->output;
           }

           break;
  }
}  /* end of SEND_OUTER_LAYER  */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F
.10----------------------------------*/

process_neuron(this_neuron)
NEURON    *this_neuron;

/* Will just process the inputs add them according to law and
   store the output so the layer is an independent block  */
{  /* begin of PROCESS_NEURON  */
    int  counter;
    double   pow(), hold_value;
```

```
hold_value = this_neuron->bias[0];
for ( counter =0; counter < this_neuron->no_of_inputs ; ++counter)
    {
      hold_value += (this_neuron->weight_list[counter]) *
                            (this_neuron->input_list[counter]);
     }
 /* Here the actual function will be include as a pointer to have any
      transfer function available instead of the sharp threshold */
    this_neuron->output = sigmoid(hold_value, this_neuron->threshold
);


}  /* end of PROCESS_NEURON  */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 1 1
----------------------------------*/

process_layer(this_layer)
LAYER   *this_layer;

{  /* begin of PROCESS_LAYER  */
    int counter;
    for( counter =0; counter < this_layer->no_of_neurons ; ++counter)
       process_neuron(this_layer->this_neuron[counter]);

}  /* end of PROCESS_LAYER  */

/ * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - F    . 1 2
----------------------------------*/

process_network(this_network)
NETWORK   *this_network;

{  /* begin of PROCESS_NETWORK  */
    int counter;

  /* start with first layer...*/
   send_outer_layer(this_network, 0);
   process_layer(this_network->this_layer[0]);
   print_layer(this_network, 0);
   if ( this_network->no_of_layers > 1)
     send_layer(this_network, 0);

   /* Now go the inner(hidden layers...if any */
   if ( this_network->no_of_layers > 1)
     for (counter=1; counter< (this_network->no_of_layers-1);
++counter)
     {
       process_layer(this_network->this_layer[counter]);
       print_layer(this_network, counter);
       send_layer(this_network, counter);
     }
```

```
  /* Finally the last layer and done ...and handle for one layer only.

*/
  if ( this_network->no_of_layers > 1)
 process_layer(this_network->this_layer[this_network->no_of_layers -
1]);
  send_outer_layer(this_network, 1  ); /* send outer anyway */
  if ( this_network->no_of_layers > 1)
    print_layer(this_network, this_network->no_of_layers -1);

}  /* end of PROCESS_NETWORK  */


process_network_b(this_network)
NETWORK   *this_network;

/* Same as previous routine..but it'll do it silently and will not
change the net_output/input_list vector on network ..used by
backpropagation train..*/

{  /* begin of PROCESS_NETWORK_B  */
    int counter;

  /* start with first layer...*/
   send_outer_layer(this_network, 0);
   process_layer(this_network->this_layer[0]);
   /* Now go the inner(hidden layers...if any */
   if ( this_network->no_of_layers > 1)
      send_layer(this_network, 0);

   if ( this_network->no_of_layers > 1)
     for (counter=1; counter< (this_network->no_of_layers-1);
++counter)
     {
       process_layer(this_network->this_layer[counter]);
       send_layer(this_network, counter);
     }

  /* Finally the last layer and done ...and handle for one layer only.

*/
  if ( this_network->no_of_layers > 1)
 process_layer(this_network->this_layer[this_network->no_of_layers -
1]);

}  /* end of PROCESS_NETWORK_B  */
```

References

Anderson C, Learning and Problem Solving with Multilayer Connectionist Systems, Ph.D. Dissertation, University of Massachusetts, Sept 1986

Anderson J, Cognitive and psychological computation with neural models, IEEE Trans. on Sys, Man, Cybernet., SMC-13, N0.5, 1983, 799-815

Anderson J, Retrieval of information from long-term memory, Science, 220, Apr 1983, 25-30

Aseo, Neural Networks-out of the Lab and into the real world,Electronic System Design, September 1987, 17

Ballard D., Modular learning in neural networks, Proceedings AAAI-87, 1987 279-284

Bertsekas D, Tsitsiklis J, Parallel and Distributed Computation, Prentice Hall, New Jersey 1989

Blelloch G, CIS: a massively concurrent rule-based system, Proc. 5th Nat. Conf. on AI, Phil. Pa, Aug 1986 735-741

Blelloch G, Rosenberg C, Network learning on the connection machine, Proc. 10th IJCAI, 1987, 323-326

Bower J, Biological Based Neural Networks, Unpublished Manuscript, 1988

Burkowski F, et. al., A Message-Based Architecture for High
Concurrency, in Hypercube Multiprocessors 1986, M. Heath (ed), SIAM, Philadelphia, 1986, 27-37

Burr D, A Neural Network Digit Recognizer, Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Oct 1986

Brown J, Parallel architectures for computer system, Physics Today, May 1984, 28-35

Bruce A, et al., Learning and memory properties in fully connected networks, Neural Networks for Computing, J. Denker (ed), American Instit. Physics, 1986, 65-70

Carpenter G, et al. Computing with neural networks, technical comments, Science, 235, 1987 1226-1229

Carpenter G, Grossberg S, Absolutely stable learning of recognition codes by a self-organizing neural network, Neural Networks for Computing, J. Denker (ed), American Instit.Physics, 1986, 77-85

Carpenter G, Grossberg S, A massively parallel architecture for a self-organizing neural pattern recognition machine, Computer Vision, Graphics, and Image Processing, 37, 1987, 54-115

Carpenter G, Grossberg S, Neural dynamics of category learning and recognition: Attention, memory consolidation, and amnesia, in Advances in Psychology, Vol 42, The Adaptive Brain I, North-Holland 1987, 239-286

Carriero N, Gelernter D, Linda on Hypercube Multicomputers, in Hypercube Multiprocessors 1986, M. Heath (ed), SIAM, Philadelphia, 1986, 45-56

Caudill M, Neural Networks Primer, Part I, AI Expert, Vol 2 No 12, 1987, 46-52

Caudill M, Neural Networks Primer, Part II, AI Expert, Vol 3 No 2,1988, 55-61

Caudill M, Neural Networks Primer, Part III, AI Expert, Vol 3 No 6, 1988, 53-59

Caudill M, Neural Networks Primer, Part IV, AI Expert, Vol3, No 8, 1988, 61-67

Caudill M, Neural Networks Primer, Part V, AI Expert, Vol 3 No 10, 1988, 57-65

Caudill M, Neural Networks Primer, Part VI, AI Expert, Vol4, No 2, 1989, 61-67

Caudill M, Neural Networks Primer, Part VII, AI Expert, Vol 4 No 5, 1989, 51-58

Chen H, et. al., High order correlation model for associative memory, Neural Networks for Computing, J. Denker (ed), AIP, 1986,

Chun H, et al, Network regions: alternative to the winner-take-all structure, Proc. 10th IJCAI, 1987, 380-387

Cohen M., Grossberg S., Absolute stability of global pattern formation and parallel memory storage by competitive neural networks, in Advances in Psychology, Vol 42, The Adaptive Brain I, North-Holland 1987, 288-308

Cottrell G., A model of lexical access of ambiguous words, Proc. Nat. Conf. AI, Austin Tx, Aug 1984, 61-67

D'Autrechy C, Reggia J, The MIRRORS/II simulator, Proc. 20th Annual Simulation Symp. March 1987, 121-132

Dell G, Positive feedback in hierarchical connectionist models: applications to language production, Cognitive Science, Vol 9, 1985, 3-23

Dimopolous N, Organization and stability of a Neural Network class and the structure of a multiprocessor system, Ph.D. Dissertation, University of Maryland, 1980

Dimopolous N, On the structure of the homogeneous multiprocessor, IEEE Trans. on Computers, C-34, No.2 1985, 141-150

Eccles J, The modular operation of the cerebral neocortex considered as the material basis of mental events, Neuroscience, Vol 6 No. 10, 1981, 1839-1856

Eich J, Levels of processing, encoding specificity, elaboration, and CHARM, Psycological Review, 92, No.1, 1985, 1-38

El-Leithy N, A semistate model for neural-type junction circuits, unpublished Univ. Maryland

Fahlman S, Hinton G, Connectionist architectures for artifical intelligence, IEEE Computer, Jan 1987, 100-109

Fahlman S, Touretzky D, van Roggan W, Cancellation in a parallel semantic network, Proc. 7th IJCAI, Vancouver BC, 1981 257-263

Feldman J, Ballard D, Connectionist models and their properties, Cognitive Science, Vol 6, 1982, 205-254

Ferry G, Networks on the brain, New Scientist, 16 July 1987, 54-58

Fukushima K, et.al., Neocognitron: A neural network model for a mechanism of visual pattern recognition, IEEE Trans. on Sys. Man, Cybernet., SMC-13, No.5, 1983, 826-834

Georgopoulos A, et. al., Neuronal population coding of movement direction, Science, 233, 1986, 1416-1419

Gilbert S, Implementing Artificial Neural Networks in Integrated Circuitry: A design proposal for Backpropagation, DTIC Tech Report TR-810, Nov 1988

Giles C, Learning and generalization in high order neural networks: An overview, unpublished

Goddard N, Lynne K, Mintz T, Rochester Connectionist Simulator, Dept of Computer Science University of Rochester, TR 233, 1987

Goles E, Vichniac G, Lyapunov functions for parallel neural networks, Neural Networks for Computing, J. Denton (ed), American Inst. Physics,1986, 165-181

Griffith J, Randomly connected networks of neurons, Mathematical Neurobiology, Academic Press, 1971, 67-75, 82-85

Grossberg S, Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors, Biol. Cybernet. 23, 1976 121-134

Grossberg S, Adaptive pattern classification and universal recoding: II. Feedback, expectation, olfaction illusions, Biol. Cybernet. 23, 1976 187-202

Heath M, The Hypercube: A Tutorial Overview in Hypercube Multiprocessors 1986, M. Heath (ed), SIAM, Philadelphia, 1986, 7-10

Hockney R, Jesshope C, Parallel Computer 2, Architecture, Programming and Algorithms, Adam Hilger, Philadelphia, 1988

Hamaker V, Vranesic Z, Zaky S, Computer Organization, McGraw-Hill, New York, 1984

Hartline D, Synetsim 3.0 Preliminary Distribution Package, Beskesy Lab Honolulu HI

Hartline D, Synetsim 3.0: A model for simulating restricted neural networks, Soc. Neurosci. Abstr., Vol 13, 1987

Heath M, The Hypercube: A tutorial overview, Hypercube Multiprocessors 1986, SIAM, Philadelphia, 1986, 7-10

Hecht-Nielsen R, Counterpropagation networks, Applied Optics, Vol 26, No 23, December 1987, 4979-4984

Hecht-Nielsen R, Neurocomputing: Picking the Human Brain, IEEE Spectrum, Vol 23, No. 3, Mar 88, 36-41

Hendler J, Marker-passing and microfeatures, Proc. 10th IJCAI, 1987, 151-154

Hillis W, The Connection machine: a computer architecture based on cellular automata, Physics 10D, 1984, 213-228

Hoffman G, Benson M, Neurons with hysteresis form a network that can learn without any changes in synaptic connection strengths, Neural Networks in Computing, J. Denton (ed), American Instit. Physics, 1986 219-226

Hopfield J, Neural networks and physical systems with emergent collective computational abilities, Proc. Natl. Acad. Sci. USA 79, Apr 1982, 2554-2558

Hopfield J, Neurons with graded response have collective computational properties like those of two-state neurons, Proc. Natl. Acad. Sci. USA 81, May 1984, 3088-3092

Hopfield J, Learning algorithms and probability distributions in feed-forward and feed-back networks, Proc. Natl. Adad. Sci. USA, Vol 84, December 1987, 8429-8433

Hopfield J, Tank D, Computing with neural circuits: A model, Science, 233, 1986, 625-633

Hopfield J, Tank D, "Neural" computation of decisions in optimization problems, Biol.Cybernet. 52, 1985, 141-152

Hoppensteadt F, Electrical models of neurons, Lectures in Applied Mathmatics Vol 19, 1981, 327-344

Howard R, et.al., An associative memory based on an electronic neural network architecture, IEEE trans. on Electronic Devices, Vol ED-14, No 7, July 1987, 1553-1556

Huang W, Lippman R, Comparisons between Neural Networks and conventional classifiers, Proceedings of IEEE First Annual International Conference on Neural Networks at San Diego, Vol IV, SOS Printing, San Diego CA, 1987, 485-494

Huberman B, Hogg T, Phase transitions in artificial intelligence systems, Artificial Intelligence 33, 1987, 155-171

Indurkhye B, et al, Optimal partitioning of randomly generated distributed programs, IEEE Transactions on Software Engineering, SE12, No 3, Mar 1986, 483-495

Jackel L, et al, Electronic Neural Networks, 1989 Digest of Papers for the Microcircuit Application Conference, Orlando FL, Nov 1989, 215-218

Jones M, Feedback as a coindexing mechanism in connectionist
architectures, Proc. 10th IJCAI, 1987, 602-610

Jones W, Hoskins J, Back-propagation, BYTE, October 1987, 155-162

Josin G, Neural-network Heuristics, three heuristic algorithms that learn from experience, BYTE, October 1987 183-192

Kabrisky M, et. al., A multiplexed multi-electrode semiconductor brain electrode implant, Presented at SMC Conference Oct 1987 Alexandria Va

Kallstrom M, et. al., Programming Three Parallel Computers, IEEE Software, January 1988 11-22

Kearns T, A Methodology, Based on Analytical Modeling, for the Design of Parallel and Distributed Architectures for Relational Database Query Processors, Ph.D. Dissertation, Air Force Institute of Technology, May 1987

Kohonen T, Self-Organization and Associative Memory, Springer-Verlag, Berlin, 1984

Kohonen T, Self-organized formation of feature maps, Cybernetic Systems, Recognition, Learning, Self-Organization, Caianello & Musso,John Wiley & Sons Inc. 1984, 3-12

Klopf A, A drive-reinforcement model of single neuron function: an alternative to the hebbian neuronal model, Neural Networks for Computing, J. Denker (ed), American Instit. Physics, 1986, 265-270

Kosko B, Constructing an Associative Memory, BYTE, Sep 1987, 137-144

Kosko B, Adaptive bidirectional associative memories, Applied Optics, Vol 26, No 23, December 1987, 4947-4960

Kruatrachue B, et. al., Grain size Determination for Parallel
Processing, IEEE Software, January 1988 23-32

Lapedes A, Nonlinear Signal Processing with Neural Networks: Prediction and System Modeling, Los Alamos National Laboratory Report LA-UR-87-2662, July 1987

Lawrie D, Access and Alignment of Data in an Array Processor, IEEE Transactions on Computers, C-24, Dec 1975, 496-503

Levin E, Analyzing Protein Structures with Neural Networks, Neural Networks for Computing Conferences; Snowbird, UT, American Institute of Physics, New York, 1986

Linsker R, From basic network principles to neural architecture: emergance of spatial-opponent cells, Proc. Natl. Acad. Sci. USA, Vol 83 Oct 1986, 7508-7512

Linsker R, From basic network principles to neural architecture: emergance of orientation-selective cells, Proc. Natl. Acad. Sci. USA, Vol 83 Nov 1986,8390-8394

Linsker R, From basic network principles to neural architecture: emergance of orientation columns, Proc. Natl. Acad. Sci. USA, Vol 83 Nov 1986, 8779-8783

Lippmann R, An introduction to computing with neural nets, IEEE ASSP, April 1987 4-22

Madore B, Freedman W., Self-organizing structures, American Scientist, Vol 75, 1987, 252-259

Malsberg C, Self-organization of orientation sensitive cells in the striate cortex, Kybernetik Vol 14, 1973, 85-100

McEliece R, et al, The capacity of the Hopfield associative memory, IEEE Trans on Information Theory, Vol IT-33, No 4, July 1987, 461-482

Meng B, Parallel Processing makes compiler advances, Electronic System Design, March 1987, 61-64

Minsky M, Papert S, Perceptrons, MIT Press 1969, 1-20, 161-170

Nicol D, Analysis of Optimal Random Program Partitions, ICASE Report No 86-53, NASA Langley Research Center, Aug 1986

Newcomb R, El-Leithy N, Semistate description of an MOS neural-type cell, unpublished Univ. Maryland

Palmer J, A VLSI Parallel Supercomputer, in Hypercube Multiprocessors 1986, M. Heath (ed), SIAM, Philadelphia, 1986, 19-26

Pazzani M, Dyer M, A comparison of concept identification in human learning and network learning with the generalized delta rule, Proc. of 10th IJCAI, 1987, 147-150

Peeling S, The Multilayer Perceptron as a Tool for Speech Pattern Processing Research, Proceedings IoA Autumn Conference on Speech and Hearing, 1986

Peretto P, Niez J, Stochastic dynamics of neural networks, IEEE Trans. on Sys, Man, Cybernet., SMC-16, No.1, 1986, 73-83

Pfister G, Norton V, "Hot Spot" Contention and Combining in Multistage Interconncetion Networks, Proceedings of the 1985 International Conference on Parallel Processing, Aug 1985

Port O, Computers that come awfully close to thinking, Business Week, 2 June 1986, 92-96

Qian N, Sejnowski T, Predicting the secondary structure of globular proteins using neural network models, Neural Networks for Computing Conference: Snowbird, UT, American Institute of Physics, New York, 1988

Reggia J, Virtual lateral inhibition in parallel activation models of associative memory, Proc. 9th IJCAI, Los Angeles Calf, Aug 1985, 244-248

Rosenblatt F, Two theorems of statistical separability in the
perceptron, Mechanization of Thought Processes, London, Her Majesty's Stationary Office, 1959, 421-450

Rumelhart D, McClelland J, Parallel Distributed Processing:
Explorations in the Microstructure of Cognition, Vol 1: Foundations, MIT Press, 1986

Rumelhart D, McClelland J, Interactive processing through spreading activation, Interactive Processing in Reading, L. Erlbaum Assoc. Inc., 1981, 37-60

Serlin O, Parallel processing: fact or fancy, Datamation, 1 Dec 1985, 93-105

Shaw G, Silverman D, Pearson J, Model of cortical organization embodying a basis for a theory of information processing and memory recall, Proc. Natl. Acad. Sci. USA, Vol 82, Apr 1985 2364-2368

Shrager J, Hogg T, Huberman B, Observation of Phase transitions in spreading activation networks, Science, Vol 236, May 1987, 1092-1094

Small S, et. al., ISCON: A Network Construction Aid and Simulator for Connectionist Models, Dept of Computer Science, University of Rochester, TR 109 April 1983

Stein J, Role of the cerebellum in the visual guidance of movement, Nature, Vol 323 Sep 1986, 217-221

Stone H, High Performance Computer Architectures, McGraw-Hill, New York, 1987

Stornetta W, Huberman B, An improved three-layer, back propagation algorithm, Proc. 1st Intl. Conf. on Neural Networks, San Diago Calf. 1987

Torras C, Neural network model with rhythm-assimilation capacity, IEEE Trans, Sys, Man, Cybernet. SMC-16 No. 5, 1986, 680-693

Torre V, Poggio T, A synaptic mechanism underlying directional selectivity to motion, Proc. R. Soc. Lond. B. 202, 1978, 409-416

Touretzky D, BoltzCONS: reconciling connectionism with the recursive nature of stacks and trees, 8th Annual Conf. of the Cognitive Science Society, Aug 1986, 1-12

Touretzky D, Hinton G, Symbols among the neurons: details of a connectionist inference architecture, Proc. 9th IJCAI Los Angeles Calf., Aug 1985, 238-243

Tseng P, A Parallelizing Complier For Distributed Memory Parallel Computers, PhD Dissertation, Carnegie Mellon University, May 1989

Wasserman P, Neural Computing Theory and Practices, Van Nostrand Reinhold, New York, 1989

Werbos P, Building and understanding adaptive systems, A statistical numerical approach to factory automation and brain research, IEEE Trans Syst Man Cybernet, SMC-17 No 1, 1987, 7-20

Widrow B, Winter R, Baxter R, Learning Phenomena in layered neural networks, IEEE Proceeding 87, 1987, Vol II 411-429

Wolfe M, Multiprocessor synchronization for concurrent loops, IEEE Software, Jan 1988, 34-42

Wolfram S, Cellular automata as models of complexity, Nature, Vol 311, Oct 1984, 419-424

Will C, Neural Network Architectures and their Implications for Next Generation Information Systems, Proc. 26th Annual Tech Symposium of ACM Wash D.C., 11 June 1987, 32-38