Analysis, Representation and Mapping of Neural Networks onto Parallel Hardware

Uğur Bilge

a thesis submitted for the degree of

Doctor of Philosophy in Computer Science

University of London

Department of Computer Science
University College London
Gower Street, London WC1E 6BT

September 1993

Abstract

Neural networks provide solutions to a class of pattern recognition and optimisation problems that are hard to solve with conventional techniques. Currently, most neural network applications are computationally intensive simulations on conventional sequential computers. As a solution general-purpose parallel architectures are increasingly used to speed up simulations. Hence, there is a growing need for generic strategies for simulating neural networks on parallel computers. This thesis investigates generic representation and mapping strategies for neural networks on general-purpose parallel architectures. The research comprises three main parts: an analysis of neural network models, an analysis of neural network representations, and by utilising these analyses, the representation and mapping of neural networks on parallel hardware.

To understand the computational and structural properties of neural network models, and to establish a generic representation, an in-depth analysis is carried out in the form of three case studies. The *Hopfield*, the *Self-Organising Map* and the *Backpropagation* models are used respectively in three appropriate real-world applications; *pattern recognition*, *data clustering* and *financial forecasting*.

Neural network representations determine parallel mapping options and the subsequent efficiency of mappings. Function-oriented, object-oriented and matrix-based representations are examined with examples, stressing their advantages and disadvantages. A matrix-based *C* library *MATLIB* and a neural network library *NETLIB* are put forward as generic, modular and flexible means to represent neural networks and exploit parallel, general-purpose execution environments.

The mapping of neural networks onto parallel hardware is a computational optimisation problem with two main constraints: *processing costs* and *communications costs*. The Mapper's task is to optimise for a fast and efficient execution, by partitioning and distributing neural network representations across a number of parallel processors, and scheduling the parallel execution. A Computational Analysis Tool (CAT) is developed to calculate processing and communications costs, and to detect parallelism in a given *MATLIB* definition. An Automatic Parallel Mapper (APM), using this analysis, can partition the representation and generate parallel or pipelined code with appropriate data exchange instructions between the parallel processing modules.

The **Esprit II Galatea General Purpose Neural Computer** (**GPNC**) is used as a test and implementation domain for this research work. The **GPNC** is a multi-processor architecture consisting of a host and a number of parallel *Virtual Machines* (**VM**), each containing a local *CPU* and a *co-processor board*, communicating and interpreting a matrix-based intermediate-level language called *VML*. The Galatea Mapper is designed and developed for semi-automatic mapping of *VML* rules to a number of parallel **VMs**.

To assess the performance of the mapping strategies, *MATLIB* definitions of the three neural network models are partitioned and simulated in parallel on a network of SUN workstations. CAT projections are used to authorise data or task parallel mappings automatically. *Multiple neural network* applications are also simulated with two or more neural networks cooperating or competing in the solution of a problem.

This thesis shows that the matrix-based abstraction captures neural network properties, and the computational cost analysis based mapping strategy is generic, flexible and can be automated. In addition, the simulation results show that: *(i)* the three neural network models studied in this thesis are tightly coupled algorithms, and are not suitable for pipeline or task parallelism, *(ii)* data parallelism for these models can increase performance only if fast communications interfaces are provided, and *(iii)* current distributed computer networks can be used for multiple neural network simulations, producing clear gains in performance.

To the memory of my mother

Acknowledgements

I would like to thank my supervisor Professor Philip Treleaven for his guidance and support throughout my research, and for always making time, whenever I needed to consult him.

I wish to thank my friend Dr. Paulo Rocha for his encouragement and support. I am grateful for his meticulous reading of this thesis and his detailed comments on it.

I also wish to thank Dr. Peter Rounce for his valued contribution in finding time to read and comment on this thesis.

I acknowledge my friends and colleagues Ing. Cesare Alippi, Jason Kingdon, Mike Hewetson, Anoop Mangat, Dr. Meyer Nigri, Michael Recce, John Taylor, Dr. Marley Vellasco and many others in the Department of Computer Science at UCL, for their various kinds of support during my research.

Last but not least, I am grateful to my wife Pippa for her patience and encouragement during the writing up of this thesis, without which the thesis would not have been completed.

Table of Contents

List of Figures

List of Tables

Chapter 1

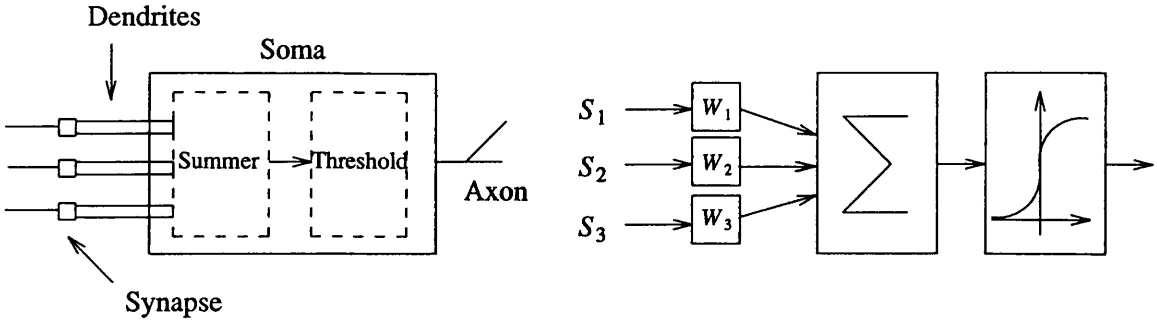
Introduction

This chapter presents a brief introduction to Neural Networks, outlines the motivations and research goals of this work, and gives an overview of the research contributions and thesis organisation.

1.1. Neural Networks

The Neural Networks field is an information processing paradigm that has been inspired by the organisation of the brain. This field has surged over the last 5 years, becoming one of the fastest growing computing technologies. As a multidisciplinary field [24, 37, 148], it covers areas such as neuroscience, psychology and computing. Neural Computing specifically deals with neural network algorithms, applications, programming and execution environments.

The essence of neural computing is to use networks of neuron-like simple Processing Elements (PE), or Artificial Neurons (AN) as computational devices [110]. Artificial Neural Networks are highly interconnected structures of artificial neurons which are modeled on an idealised view of biological neurons. A biological neuron, as is well known, is the basic building block of the nervous system, consisting of a cell body, branching extensions called *dendrites* which receive signals, and an *axon* which passes the neuron's output to the other neurons (Figure 1.1). The junctions between axons and dendrites are called *synapses*. To oversimplify somewhat, a neuron collects the signals from its synapses, and sums them. If the combined strength of the signals exceeds a certain limit - a threshold, the neuron sends out a signal through its axon. These steps are electrochemical operations carried out by over fifty different neurotransmitters, involving many different neuron structures. Neurons typically take a millisecond to respond to their inputs; in the same amount of time, conventional computers can carry out millions of calculations. Yet the biological network of neurons is much faster in pattern recognition tasks than conventional computers. There are approximately 100 billion neurons in the human brain, and one neuron may be connected up to 10,000 others; thus the main strength of the neural model comes from this massive interconnectivity and parallelism.



(a) Biological Neuron (b) Artificial Neuron

Figure 1.1. Models of Neurons

An artificial neuron imitates a biological neuron in three aspects: it has weighted input connections, a summation function and a threshold function that generates the output. To further the similarity in an attempt to build a replica of the brain [101] is not yet feasible, for two reasons. Firstly, our understanding of the brain is limited, and secondly, today's technology cannot match the same level of processor interconnectivity on silicon circuits. Despite the high level of interconnectivity, the brain is not homogeneous. Computations in the brain seem to be localised for specific tasks. PET (Positron Emission Tomography) scans and MRI (Magnetic Resonance Imaging) results confirm that the brain has a modular structure. These results encourage the design of modular networks and distributed parallel hardware platforms, to overcome the interconnectivity bottleneck.

	The Brain	**Artificial Neural Networks**
Organisation	network of neurons	network of Processing Elements
Components	dendrites & axon synapses	inputs & output weights
Processing	analog	digital or analog
Architecture	10-100 billion neurons	1-1,000,000 PEs
Interconnectivity	1 neuron to 10,000 others	limited and slow
Hardware	neuron	switching device
Switching Speed	1 millisec	1 nanosec - 1 millisec
Technology	biochemical	silicon, optical

Table 1.1. The Brain and Neural Networks

Today, most neural network applications are software simulations, running on conventional, sequential computers. Some simulations run on parallel hardware, and some hardware implementations already exist [143]. Neural network simulations are computationally demanding, requiring high performance hardware platforms with large memories. Figure 1.2 shows the computational requirements for a range of neural network applications and the performance of neural network simulators (CPS stands for Connections Per Second; a performance criterion which is widely used.) As can be seen, using parallelism, both the speed and the storage capacity can be improved. One major problem is to program or map software onto parallel hardware, and exploit parallel execution environments efficiently. This thesis specifically deals with the mapping of neural networks onto parallel hardware for efficient execution. Four main research areas of neural computing are considered in the course of this research. These are neural network models and applications, programming and execution environments.

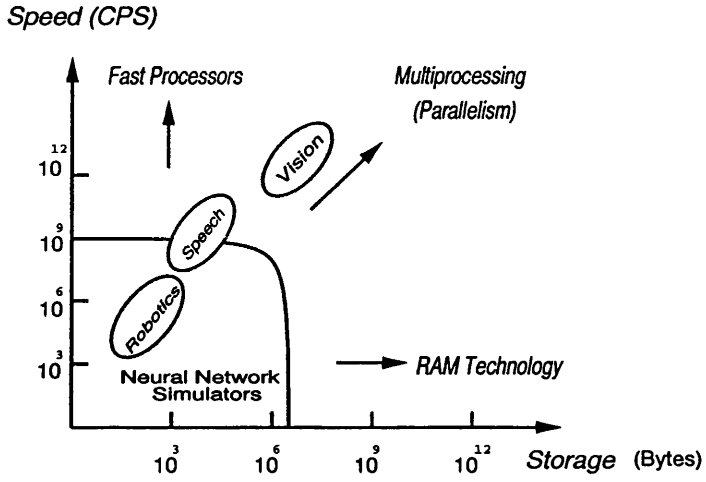


Figure 1.2. Applications and Simulators [6]

Neural Network Models and Applications

A large number of models have been developed and tested on a number of real-world applications. Models differ from one another by the interconnection topology and the varying properties of artificial neurons. Today, there are over 100 established network models [99, 144]. The best known models are the Hopfield nets [72], the Self-Organising Map (SOM) [88], and the Backpropagation model [130]. These three models are analysed in detail in three case studies in chapter 4.

Most neural network algorithms are mathematical and statistical models of learning. They achieve learning through an iterative process called the *training* phase. In training, a network adaptively modifies the interconnection weights between the Processing Elements, following a learning rule. *Recall*, on the other hand, involves using the weight space after training, to produce output values for given input patterns.

Neural networks are suitable for a diversity of real-world applications covering brain modelling, speech and image processing, robotics and control, planning and optimisation, and human-machine interfacing systems [129]. The main applications are pattern recognition problems which are difficult to solve by conventional techniques. Other popular applications are optimisation problems which take an impractical amount of computer time to reach a solution using conventional techniques. In the last 3 years, financial applications have gained momentum. As part of wider data processing applications, neural networks are used in predicting trends in prices, stock market and exchange rate forecasting [86], credit and insurance assessments [34]. An interesting image processing application is the recognition of hand-written characters. This has become particularly important in the wake of pen driven computers [22, 57], an important step in the human-machine interfacing systems.

Neural networks present different computational requirements during the training and recall phases, depending on applications. A neural network study [6] outlines these requirements as memory (connections) and execution speed (connections per second). For a robotic arm manipulator, real-time speed requirements are estimated as 350,000 CPS, and the computational demand in vision applications can go up to 2 GCPS. In most cases training can be done off-line, but recalls often require a real-time performance. These requirements are still far from satisfactory, presenting a stumbling block in the neural network development.

Neural network research is recently focusing on modular, hybrid systems. These systems require coarse-grained parallel distributed architectures consisting of general-purpose high performance modules. Mapping and execution strategies must consider these modular trends in models and applications.

Neural Network Programming and Execution

Most neural network applications are developed as computer simulations, and some models are realised as hardware implementations. A neural network simulation is developed in the following way: first, a suitable model is selected and adapted for the application, a suitable hardware platform is chosen, a computer program of the model is written, and finally the simulation is executed on the hardware. This process involves two separate domains, the software domain and the hardware domain. Research in Programming Environments deals with issues relating to software, such as the user interface, network representations and mapping. Research in Execution Environments on the other hand deals with hardware matters for the efficient execution of applications.

Neural Network Programming Environments (NNPE) are sophisticated programs which ease the cycle of program development and execution of an application. They facilitate the transformation of real-world problems onto computer programs with user-friendly interfaces and testing, debugging and monitoring functions. Most NNPEs contain the following modules: a graphics monitor and a high-level language for user-friendly definition of networks, an intermediate-level neural network specification language for low level representation of the applications, and an algorithm library consisting of parameterised models. A number of research systems and commercial products have been developed in this field, and they are reviewed in chapter 2.

Neural Network Execution Environments are high performance hardware platforms where neural network execution takes place. Today, most execution environments are sequential computers. For small applications where training can be done off-line, PCs are often sufficient. Early PCs delivered 25,000 CPS execution speeds [6], and this figure has been increased tenfold in the last 5 years. Neural network applications such as signal processing and vision tasks require vast amounts of processing power. Neurocomputers are developed to provide this high performance. communications facilities. Neurocomputer research is progressing in two directions to overcome these difficulties.

Special-purpose Neurocomputers are usually fine-grained compact VLSI implementations of certain neural network models. The Hopfield nets and the Self-Organising Map [12,90,101,102] are the two most popular models implemented in hardware because of their simplicity. Special-purpose neurocomputers provide fast execution (currently 10^9 CPS), and can be implemented as compact analog circuits, but they lack flexibility and programmability.

General-purpose Neurocomputers on the other hand are general, flexible and programmable platforms, but they provide poorer performance rates (typically in the range of $10^7 - 10^8$ CPS) [107,120]. They are often coarse-grained architectures aiming at high performance and generality. Most neural network models and some non-neural network applications can be executed on these machines. The general-purpose neurocomputers are mainly digital circuits with some analog hybrid implementations. A detailed review of the current hardware platforms is presented in chapter 2.

Considering neural networks' inherent parallel nature, their potential can be fully exploited on massively parallel architectures. Yet, there are practical difficulties in implementing highly interconnected massively parallel architectures with efficient These difficulties can be overcome by parallel, distributed and modular designs. Future architectures can make use of special and general purpose neurocomputers by combining them in a parallel distributed framework. These systems will be able to incorporate multi-domain applications and hybrid solutions, and run on heterogeneous architectures, consisting of general-purpose and dedicated processors.

Mapping Neural Networks onto Parallel Hardware

The rapid development of computer hardware in speed, memory and data storage facilities has not been sufficient to meet the demand for high performance computing in some applications. Neural network applications are among these applications which are computationally intensive on conventional computer simulations. Given the choice of having a high performance system now or waiting five years, parallelism can deliver the required performance by exploiting current hardware platforms [121]. In fact, parallelism increasingly is used in the design of single-chip microprocessors. Yet parallelism introduces one important issue; the programming or mapping of applications onto hardware. Mapping involves decomposing and partitioning representations and distributing them onto parallel processors for a fast and efficient execution. Neural networks are intrinsically parallel algorithms, and it is natural to think they can be executed most efficiently on parallel architectures. In reality, inefficient use of parallel resources can result in slower execution rates over sequential executions.

A parallel neural network simulation is closely linked to the application and model which are mapped, and to the programming and execution environments where simulation and finally execution takes place. Overall, the efficiency of execution on a parallel system depends on the following factors:

- *Efficient representation* - The neural network specification language must be general, flexible and allow easy manipulation by the mapping process.
- *Efficient mapping* - The mapper must efficiently partition and distribute the neural network representation across a number of parallel processors.
- *Processor speed* - The speed of the processors in the system directly affects the speed of the execution.
- *Processor memory* - The size of memory and the access time are important, as neural network concepts, such as weights, patterns, and look-up tables are stored in Random Access Memory.

- *Interprocessor communications* - Interprocessor communications facilities must be sufficient both in speed and latency to cope with the data transfer and update throughout the network.

1.2. Research Goals

The main goal of this research is to investigate generic strategies for mapping neural networks onto parallel hardware. Part of this goal is to achieve a generic representation that captures neural network properties and facilitates parallel mapping. The following are the main criteria to be met in the research for representation and mapping strategies in this thesis:

- **High Performance**: Execution speed must be improved over sequential speed as a result of the mapping.
- **Generality**: The system must provide high performance for a wide range of neural network models and applications, and support a variety of neurocomputers.
- **Flexibility**: The system must be easily modified, and it must allow for expansion for new applications, models, and hardware platforms.
- **Modularity**: The system must be modular to allow for expandability and upgradability.
- **Efficiency**: The system must supply efficient use of resources.
- **Scalability**: The system must scale up to increasing number of processors.
- **Automation**: Parallel code should be generated and distributed automatically.

This thesis attempts to achieve these research goals within the framework of a General Purpose Neural Computer (GPNC). A GPNC is a high performance system, consisting of a coarse number of parallel general-purpose modules, all connected to a host computer with a user-friendly, flexible programming environment, capable of executing a wide range of neural network models and applications (Figure 1.3).

The ESPRIT II Galatea GPNC is a typical example of such a system which is potentially capable of delivering a high performance for a wide range of neural network models and applications. The Galatea GPNC is currently under development, and it has been used as the test and implementation domain for mapping and scheduling strategies developed in this thesis. The design and development of the Galatea Mapper and the

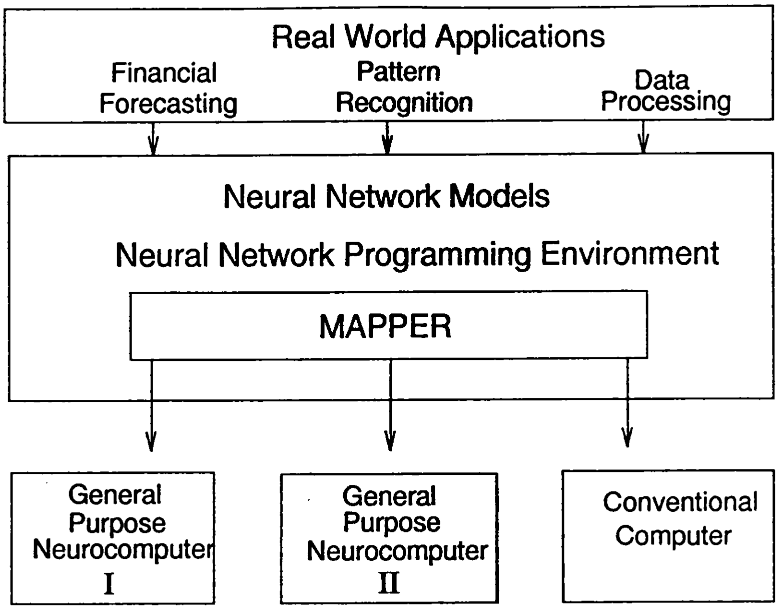


Figure 1.3. General Purpose Neural Computer and Mapper

specification of the Scheduler are parts of the research work undertaken during this thesis research.

1.3. Research Plan

To achieve the research goals;

- An in-depth analysis of neural networks has been carried out in the form of three case studies. The Hopfield nets, the Self-Organising Maps and the Backpropagation model with three appropriate applications have been used in the case studies. These three models are selected because they represent a good cross-section of most neural network models in terms of their applications, structures, training and recall procedures. This analysis aims to : *(i)* understand neural network models and their computational properties, *(ii)* highlight suitable application domains, *(iii)* explore potential structural parallelism, and *(iv)* search for a generic representation.

- A comparative analysis of neural network representations has been carried out, using function-oriented, object-oriented and vector-oriented representations in a number of simulations. The analysis aims to establish a representation strategy which is capable of *(i)* capturing most neural network properties, and *(ii)* exploiting general-purpose parallel hardware. As a result a matrix-based library *MATLIB* and a neural network library *NETLIB* have been put forward.

- The mapping strategy has been outlined as a computational optimisation process which is generic, flexible and upgradable. A Computational Analysis Tool (CAT) and an Automatic Parallel Mapper (APM) have been developed for automatically partitioning and mapping *MATLIB* representations onto parallel processors.

- The Galatea Mapper has been implemented as part of the programming system for the Galatea GPNC simulator, and is used for demonstrating mapping and scheduling strategies. *VML*, the intermediate-level language of the Galatea GPNC, has been semi-automatically partitioned and parallel executions are simulated on a network of SUN workstations.

- *MATLIB* representations have been partitioned and parallelised on a network of SUN workstations to assess data and task parallel mapping strategies. CAT parallel projections are used to detect parallelism in *MATLIB* definitions of neural network models. Both single domain and multiple neural network models have been used in these parallel simulations.

1.4. Research Contributions

The main contributions of this work to Neural Computing are:

- **The analysis of neural networks.** An analysis of three neural network models with appropriate real-world applications, in the form of three case studies.

- **The analysis of neural network representations.** A comparative analysis of function-oriented, object-oriented and vector-oriented neural network representations.

- **The design and implementation of *MATLIB*.** A matrix-based *C* library with parallel features. *MATLIB* is a generic representation domain for neural networks and is suitable for mapping onto general-purpose parallel hardwares.

- **The design and implementation of *NETLIB*.** *NETLIB* is a parallel neural network library with parameterised routines for the training and the recall functions of the three models, exploiting *MATLIB* functions.

- **The design and implementation of the CAT.** CAT provides a computational profile for *MATLIB* programs and detects parallelism in these representations.

- **The design and implementation of the APM.** APM exploits the computational profile provided by CAT, and automatically partitions and generates parallel or pipelined

MATLIB code.

- **The design and implementation of the Galatea Mapper.** It provides semi-automatic mapping of intermediate-level language *VML* onto a number of parallel Virtual Machines.

- Parallel to this research, original research contributions have been made by proposing novel solutions to problems such as data clustering [26], dataset pre-processing [126], financial forecasting [27] and navigation [28]. These efforts are documented as published papers and as departmental research notes. The Galatea GPNC development is also documented in chapter 3 and chapter 7, and in the form of technical reports.

1.5. Thesis Organisation

This thesis is organised as follows:

Chapter 2 presents a survey of neural computing. An overview of neural network models and applications is given, and neural network programming and execution environments are investigated.

Chapter 3 presents the Pygmalion and Galatea programming environments as major development works in European Neurocomputing. The Galatea General Purpose Neural Computer is presented as it forms the main test and implementation domain for this thesis work.

Chapter 4 comprises an in-depth analysis of neural network models in three case studies with real-world applications. The Hopfield nets, the Backpropagation model and the Self-Organising Map are chosen as they provide a good cross-section of neural network models and applications.

Chapter 5 discusses neural network representation and programming issues. Function-oriented, object-oriented and vector-oriented approaches are compared with simulation examples. A matrix-based *C* library, *MATLIB* is put forward as a generic and flexible environment to represent neural networks and exploit general-purpose parallel hardware. A neural network library *NETLIB* is also presented in this chapter.

Chapter 6 outlines the mapping strategy as computational optimisation. Different mapping techniques are discussed, data and task parallelism costs are parameterised. The Computational Analysis Tool and the Automatic Parallel Mapper are presented for the automatic generation of parallel code.

Chapter 7 describes the Galatea Mapper implementation, as part of the Galatea GPNC development. The Galatea Mapper and the Scheduler are defined, and the implementation steps are presented. Results of parallel mappings of *VML* are presented on the GPNC simulator running on a network of SUN workstations.

Chapter 8 presents the simulation results of mapping *MATLIB* representations. Parallel simulations involving single-domain, and multi-neural-network applications are used to assess the performance of the mapping strategy.

Chapter 9 provides an assessment of the work in terms of strategy, design, implementation and results. A target review is made, and neural network model, representation and execution issues are raised.

Chapter 10 contains major conclusions of this research and discusses future directions.

Chapter 2

Neural Computing

This chapter provides a brief survey of the Neural Computing field, aiming to place the thesis work in this context. With this purpose, popular neural network models and applications, programming and execution environments are reviewed and assessed.

2.1. Introduction

Conventional Artificial Intelligence (AI) techniques, manifested as the 5th generation computers, have been successfully applied to problems where a well defined number of rules determine a system's behaviour. In the late 1970's, a number of AI-based medical diagnosis packages and chess playing programs were developed. They were so successful that the chess playing programs manage to beat over 99 % of all players. But the same AI techniques are found inadequate for problems such as speech and image recognition. There are two main reasons for this. Firstly, the rules and parameters involved in these problems are unknown or not well defined. Secondly, most pattern recognition and optimisation problems are so called NP-complete, and the computational requirements for these problems become unsurmountable as the number of parameters increases. As a result, the computational requirements increase rapidly, pushing sequential, conventional computers to the limits of their performance. Many believe the answer is in parallel processing and in new forms of computation based on parallel computing.

Neural networks, seen as part of the 6th generation computers paradigm [137] offer a radically different type of computation from conventional computers; neural computing is not rule based, and it is intrinsically parallel. But, neural computing is not a candidate to replace conventional computing, which is quite successful for symbolic and numeric applications. Instead, it complements the conventional computing systems, by providing solutions to a variety of real-world problems that involve extracting useful information from complex, noisy or uncertain data [129]. Neural networks have already been applied to a range of pattern recognition problems as sequential simulations, performing better than the conventional techniques.

The proliferating applications and models are pushing the demand for high performance simulators. As mentioned above parallelism can be one way of speeding up computationally intensive simulations. Yet it brings together difficulties of parallel programming which is the topic of this research. In the pursuit of generic representation and mapping strategies of neural networks this chapter surveys the state of the art neural network algorithms, applications, programming and execution environments.

2.2. Applications and Algorithms

Popular Applications

Constructing a taxonomy of neural network applications is not easy because the same application can be categorised as an optimisation, a pattern classification and a pattern recognition task, depending on the point of view. Although the boundary is not so clear-cut, most neural network applications are optimisation or pattern classification problems.

Optimisation tasks are one of the first set of complex problems neural networks addressed. Hopfield and Tank [74] showed that a neural network can find a good solution to the Travelling Salesman Problem within an acceptable time. TSP requires the shortest tour of several cities with the condition that each city is visited only once. TSP is a typical NP-complete problem; since the number of cities increases finding a solution takes an impractically long time, as the computational requirements increase exponentially. Using the Hopfield net, a representative weight space can be constructed with the connection weights between the neurons, and this space can be used to solve optimisation problems.

Pattern classification is a major application domain for neural networks, as most highly dimensional, complex real-world problems can be reduced to pattern classification problems. Traditionally, statistical classifiers are used to solve these problems [99]. These classifiers separate given patterns to a number of classes by calculating the similarity scores for each class, and selecting the closest class as the class identity of the pattern.

The main difficulty with traditional classifiers is that they are non-adaptive. Neural networks with a different internal computational mechanism, can be used as pattern classifiers. In addition to an enhanced performance in classification, they are adaptive, robust and fault tolerant. The adaptive behaviour is a direct consequence of the feedback mechanisms which most neural networks possess. In training, a neural network continuously monitors its own performance, and adapts its weights accordingly. In recall, the weights are used to establish the class identity of a novel pattern.

The Perceptron, a supervised neural network classifier, strongly resembles the traditional Gaussian classifier. In fact, the Perceptron architecture can be used as a traditional classifier with the added advantage of a potential parallel execution.

Neural networks can also be trained by using unsupervised techniques. Unsupervised neural network classifiers such as Kohonen's Self-Organising Maps are also known as clustering algorithms. These models do not require labelled data; they classify input patterns into a pre-defined number of unlabelled classes. The SOM is similar to the traditional K-means algorithm [36, 39], yet it is adaptive, and can be executed in a massively parallel fashion.

A number of real-world applications are important as part of the broad class of pattern recognition problems. These are; speech and image processing, handwritten character recognition, and sonar/radar signal processing applications. Financial forecasting is one of the areas that has boomed in the last couple of years. The high dimensional and fuzzy nature of this area make it a suitable application for neural networks.

Adaptive control is another area in which neural networks are used successfully. By classifying world data adaptively and taking an appropriate set of actions, multi-dimensional control problems can be solved. Neural network based adaptive control systems find applications in industrial robotics, manufacturing, process control and autonomous robot navigation.

Popular Models

A number of neural network models have been put forward, with various topologies, learning and recall rules. Learning rules for neural networks are so important that, sometimes the networks are named after the learning rule they use. This is the case for "Backpropagation" which is a learning rule for multi-layered architectures. Most learning rules are derived from the well known Hebbian rule. Examples of these are the outer product rule, the Delta rule, the generalised Delta rule (Backpropagation model) [84, 130]. A number of learning rules also based on the Hebbian rule, are called competitive learning rules. Typical examples are Grossberg's *instar* and *outstar*; similarly, Fukushima's Neocognitron and Kohonen's SOM use the competitive learning rules in training. The three most popular models the Hopfield nets, the Backpropagation model and the Self-Organising Maps are analysed in detail, in chapter 4. The Neocognitron, Counter-Propagation, Adaptive Resonance Theory [33], and the Boltzmann machine [65] are amongst other well known models. The Neocognitron and the Counter-Propagation are briefly described here.

The **Cognitron** and **Neocognitron** were put forward by Fukushima as close imitations of the human visual cortex [47-49]. Biological plausibility is the first objective in the design of these models. The structure of the Neocognitron is a multi-layered architecture containing excitatory and inhibitory neurons and synapses with a competitive learning mechanism. The system is a powerful image recognition system, working under the conditions of translation, rotation and distortion of the objects, even able to recreate the original image from reverse propagation of the network. The Neocognitron is fault tolerant and self-recovering when faults occur on processors. One problem with the early models was the amount of computational power required.

The **Counter-Propagation Network** (CPN) [60, 61] is a combination of two neural network models complementing each other with different properties. It is a three layered network; the first layer is the input layer where the external inputs are clamped, the second is a Kohonen layer which associates certain neurons with certain input patterns, and the third layer is a Grossberg layer following Grossberg's outstar learning rule. The resulting network can be rapidly trained and used as an approximate vector mapping system. An average training session is 100 times shorter than that required for the Backpropagation network, but CPN is not as accurate as the Backpropagation. The most interesting application for CPN is data compression. This involves splitting the CPN into two, and using the Kohonen layer as a vector quantiser. Then the labels of the quantised vectors, which are much more compact than the original vectors are transmitted and new vectors are reconstructed on an identically trained CPN system. CPN is also an interesting example of using networks as components. Similar modular neural network architectures are put forward and simulated on parallel processors in the following sections of this thesis, as powerful problem solving domains.

Conclusion

Computational requirements for neural networks are application and model dependent. Neural network representation and execution environments must meet these requirements. Current models and applications display a wide variety, and continue to expand into new domains. Table 2.1 compares popular neural network models, with their popular applications, strengths and weaknesses.

Models	Popular Applications	Strengths	Weaknesses	Remarks
Perceptrons	Pattern classification	Simple, adaptive	cannot classify complex patterns	first neural networks
Hopfield Nets	Optimisation, Planning Content-Addressable Memory	Large scale analog implementation	No learning rule Limited storage	Covergence in recall
Self-Organising Maps	Feature extraction	Statistical Analysis tool	Noise sensitive	Unsupervised learning
Backpropagation learning on MLP	Wide applications Speech, image processing	Robust Fault-tolerant	Computationally demanding	Most popular model
Counter-Propagation	Data Compression	Quick and approximate	less accurate than Backpropagation	Inverse pattern generation
Neocognitron	Hand-written character recognition	Can handle complex patterns	Computationally demanding	modeled on visual cortex

Table 2.1. Neural Networks Applications and Properties

In chapter 4, three case studies are used to assess computational requirements for three neural network models. In the remainder of this chapter Neural Network Programming Environments and Execution Environments are reviewed.

2.3. Programming Environments

Neural Network Programming Environments are software tools for developing computer simulations of real-world applications and monitoring the execution of these applications. The effectiveness of the representation and efficiency of the execution greatly depend on the programming environment. Neural network representations are crucial for mapping purposes, as they are the very domain where partitioning takes place.

NNPEs address a wide variety of users ranging from novices to experts whose purpose are research, experimentation, education or a solution to an industrial or business oriented problem. NNPEs can be classified into three categories [144]:

- *application-oriented,*
- *algorithm-oriented,* and
- *general programming environments.*

Application-oriented systems are designed to provide solutions in a certain domain of expertise. These systems are customised, user-friendly, menu-driven, and are often supported by dedicated hardware, offering little flexibility. Nestor provides one of these systems, **NLS** (Nestor Learning System), for applications covering mortgage underwriting, and risk assessment in automobile insurance [4].

Algorithm-oriented systems either provide one algorithm or they provide a number of parameterised algorithms as an algorithm library. Many programming systems provide the Backpropagation model and its variations, as it is the most widely used neural network model. Examples are the **Brainmaker** system, supplied by California Scientific Software and the **Owl** system which provides an algorithm library for computer professionals. It is based on a generic, *C* data structure encompassing 19 parameterised neural network models. Users can specify network parameters and call the library functions in their routines. The system also offers an optional graphics interface. Esprit II **Pygmalion** project [14, 15, 18, 146] also provides a *C* library of the most popular models. Applications can be developed by calling appropriate library routines with correct parameters.

General programming environments aim to address a wide variety of applications, algorithms and users. These are often research or educational systems aiming at generality and flexibility as opposed to high performance. Most generic programming environments contain the following functional modules:

- **User Interface** - This module provides flexible, modular and user-friendly access to neural network models. A number of parameterised models and a High-Level Language are provided to ease the programming process. Usually, menu-driven graphical systems support the user interface. The execution of applications can be monitored, interrupted, saved or loaded. Debugging facilities aid the monitoring in the run-time and post-mortem. Results of the execution are displayed both numerically and graphically.

- **Neural Network Representation** - This constitutes a generic, machine-independent, intermediate-level, sophisticated network specification language which is capable of covering all possible models and configurations. A number of parameterised models can also be stored as an Intermediate-Level Language Library. Advanced users can program in ILL if they want to exploit the facilities of the low level representation for a high performance execution.

- **Mapper/Translator/Compiler** - A sophisticated program or a set of programs that generate code for a variety of target hardware architectures. This can be done either by direct translation or compilation, from the intermediate-level neural network specification language to the target languages. If the target machines are parallel, the intermediate-level representation is first partitioned, then distributed over the parallel processors. A fast, efficient and optimised execution is the purpose of this module.

Popular Systems

An early example for generic programming environments came from SAIC: **GINNI**, Generic Interactive Neural Network Interpreter [2] system provided an interactive developmental system for experimentation with neural network modeling. This system contains all functional modules mentioned above: networks are defined as hierarchical data structures, a high level neural network description language is used to define new algorithms, and the networks can be executed on remote machines over the Local Area Network (LAN) using an integrated message passing system.

AXON, launched by HNC, is a machine-independent neural network specification language [53]. It combines common features of *C* and *Pascal*, and a set of neural network specific features. AXON encompasses a generic model of neural networks regardless of the topology or functionality of the artificial neurons. An object-oriented programming approach has been adopted in the network description. A generalised Processing Element concept has been adopted to allow future models to be coded in the language. The AXON Processing Element is the basic building block of all neural models. The PE contains the following attributes: Output State, Transfer Function, Connection Classes, Connection Weights, and Local Data Memory. Processing elements with different attributes can also be defined and used in the same network description.

PDP simulation package is offered to accompany the book "Explorations in Parallel Distributing Processing" [1]. It contains a set of programs written in *C*, and runs on both MSDOS and Unix operating systems. Seven algorithms are provided which included the Backpropagation, Adaptive Resonance Theory and Competitive Learning models. The system is portable, easy to use with the support of the examples in the book, but it lacks a graphics user interface.

The **Rochester** Simulator, developed by the Department of Computer Science at the University of Rochester [52], is also written in *C* and provides a data structure which defines the lowest level processing unit. The data structure is flexible, allowing the user to define various levels of networks and connections. As, only the Backpropagation model is provided with the package, users have to develop their applications in *C* using the provided template.

The **Genesis** system was developed by the Division of Computational and Neural Systems of the California Institute of Technology [151]. It promotes biologically plausible neural network modelling. The package is a Unix/C based environment representing the networks as hierarchical objects. The graphic interface Xodus uses the X Windows environment.

Conclusion

Popular neural network programming environments presented above, are mainly for research, development and experimentation. Most systems are written in *C* and provide data structures for representing neural network concepts. Systems which provide a set of simple library functions with their source code are favoured as *open systems* where users can modify and tailor the system to their specific needs. In chapter 3, the Pygmalion programming environment is presented in detail as an early system which provided a general programming environment with an objective of a parallel execution. In chapter 5, a number of representation techniques are assessed with simulation examples, and a matrix-based library is put forward as modular, clear means for neural network programming.

2.4. Execution Environments

Neural network simulations are computationally intensive, involving repetitive multiply and add operations. Real-world applications may require the processing of huge amounts of data in a short time. Depending on the application size and domain, large amounts of memory and high speed processors may be necessary.

The general trend in computing is to use parallel architectures to overcome the computational limitations of single processor systems. Indeed, recently, parallelism has been used in the implementation of single chip microprocessors [121]. Architectures such as Intel 80860 and recent RISC architectures exploit *microparallelism* to execute multiple processes simultaneously by pipelining sequential operations.

Flynn's classification envisions four types of parallelism [145]. Two of these; Single Instruction Multiple Data stream (SIMD) and Multiple Instruction Multiple Data stream (MIMD) architectures are relevant for neural network execution. SIMD Machines are synchronous, compact, usually fine-grained machines offering high performance with simple processors. MIMD Machines are asynchronous, programmable, medium or coarse-grained machines with distributed or shared memory schemes. Distributed memory (multicomputer) architectures are scalable systems, but they suffer from interprocessor data dependency and consistency problems and parallel programming difficulties. Shared memory systems are easier to program but the number of processors must be kept to a minimum because of interprocessor competition for memory access and cache coherency problems [121]. The most advanced form of MIMD machines do not have a central controller, and are fully distributed, asynchronous systems [77].

Parallel implementation is the natural method for neural networks, which are inherently parallel. There have been some special-purpose implementations of neural models. These are usually SIMD architectures with simple analog processors connected in a matrix topology [101]. For general-purpose neural computing, computational arrays provide a good execution platform [125].

Parallel or sequential, execution environments for neural networks are called *neurocomputers*. Neurocomputers appear in great varieties. On one extreme, there are dedicated parallel SIMD machines, while on the other, there are simple RAM chips such as in the WISARD system [10, 11]. Figure 2.1 shows the trade-off between performance and flexibility for a number of hardware platforms which are used as execution environments for neural network applications.

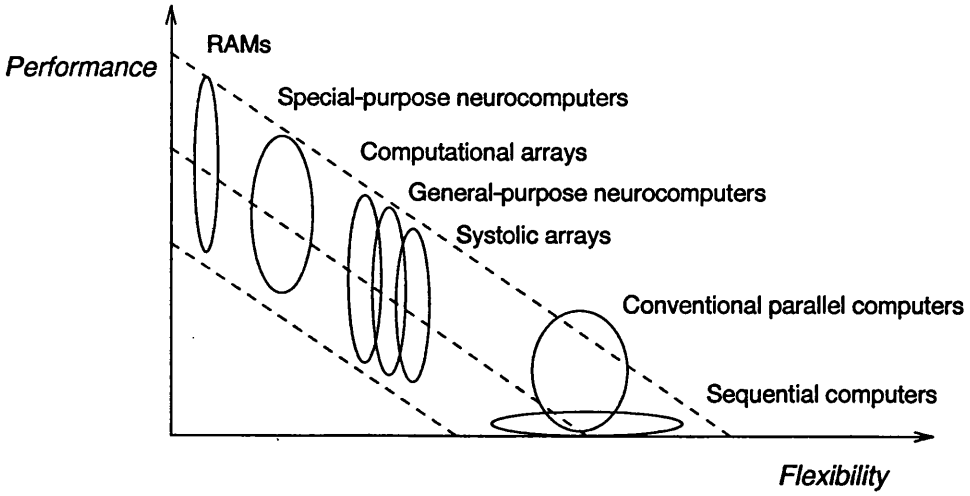

Performance
RAMs
Special-purpose neurocomputers
Computational arrays
General-purpose neurocomputers
Systolic arrays
Conventional parallel computers
Sequential computers
Flexibility

Figure 2.1. The Spectrum of Execution Environments [144]

A broad classification is possible depending on the generality and programmability of the neurocomputers. Special-purpose neurocomputers provide high performance emulating a neural net model, usually with an application in mind. They are hard-wired;

they lack flexibility and programmability. General-purpose neurocomputers can be programmed to emulate a wide range of neural net models. Some architectures can also be used for other computationally intensive tasks. Although general-purpose neurocomputers can offer a higher performance than conventional computers, they are usually a magnitude of order slower than special-purpose neurocomputers [143, 144].

2.4.1. Special Purpose Neurocomputers

Special-purpose neurocomputers are physical implementations of neural networks for high performance. The implementations are usually algorithm and application-oriented. The Hopfield nets and Kohonen's Self-Organising Maps are the two most popular models which were implemented as VLSI chips [90]. Most special-purpose neurocomputers are research systems, used in the investigation of hardware implementations for a number of neural network models and applications. The implementations of special-purpose architectures are analog, digital or optical.

Analog implementations imitate artificial neurons by using basic circuits such as transistors, operational amplifiers and resistors. The resistors are used as neural connection weights, and the amplifiers emulate the processing elements which carry out summations and apply the thresholds. Capacitors are also used to allow weight updating for on-chip learning. Special-purpose neurochips have been developed by Mitsubishi, AT&T Bell Laboratories, and California institute of Technology. These chips provide a very high performance, but they lack flexibility and programmability.

Optical implementations of special-purpose neurocomputers aim to overcome the difficulties of realising high level interconnectivity on VLSI architectures. These architectures use holograms to communicate incoming signals to a great number of processing elements. With a single hologram it could be possible to connect 10,000 light sources to 10,000 light sensors without interference even though the light beams cross each other. Optical technology could meet the interconnectivity requirements set by the biological systems. However, the technology is still young and optical neural computers are still in the early stages of development [59].

Digital systems offer the greatest flexibility, programmability and expandability. The majority of the neurocomputer implementations are digital. Some of these machines are radically different from each other, but most of them are arrays of simple parallel processors that operate concurrently. Node complexity, cost, communications facilities, granularity, parallelism and performance are important considerations in implementation.

Two examples can be given for digital general-purpose systems:

WISARD has been developed by I. Alexander's group at Imperial College, London. This system uses Random Access Memories (RAM) as neuron-like devices. The hardware implementation of the system is used in image recognition, it has shown some success in real-time face recognition.

PNNP, Probabilistic Neural Network Processor by Lockheed, is another digital special-purpose neurocomputer implementation. This system implements real-time learning for the Probabilistic Neural Network (PNN) model. The system is connected to the backplane of an IBM-PC compatible computer through a dual port memory.

Digital/Analog hybrid implementations and wafer scale integration techniques have recently gained momentum. In Japan a series of hybrid neurochips have been developed [66], and an example is Hitachi's 576 neurons Wafer Scale Integration neurochip [153].

Institut National Polytechnique de Grenoble (INPG) is also developing a neurochip which can be potentially generated in wafer scale [118]. INPG's design is a *neuron-based* architecture for executing a variety of neural network models. One goal of this project is to develop a framework for the automatic generation of application-specific VLSI chips, namely silicon compilation. The architecture is based on a processing element which is capable of performing all operations required, during the training or recall phase of a neural model. Processing elements are linked in a two-dimensional array, with each PE is connected to two external buses.

Another attempt to emulate the functionality of a neuron in hardware took place at UCL, in the **UCL Generic Neuron** project [147]. This project developed a framework to generate special-purpose chips for most neural network models The project aimed to achieve a high performance with some flexibility. A simple processing element (Figure 2.2) has been designed which can be replicated cheaply, and the functionality of this processor can be defined by users. In fact, a generic neuron is the physical implementation of the Pygmalion *nC* data structure *neuron* the Pygmalion Project is presented in chapter 3. *nC* is used to specify the connectivity and functionality of the generic neurons. The architecture uses the bus interconnection strategy for data communications, providing the system with flexibility, expandability and scalability. Any desired topology can be accomplished using the bus communications, and the access to the bus by the PEs is organised by a central controller. To reduce the communications between the PEs, both PEs sharing a connection hold copy of weight, but this would increase the memory requirements on the chip.

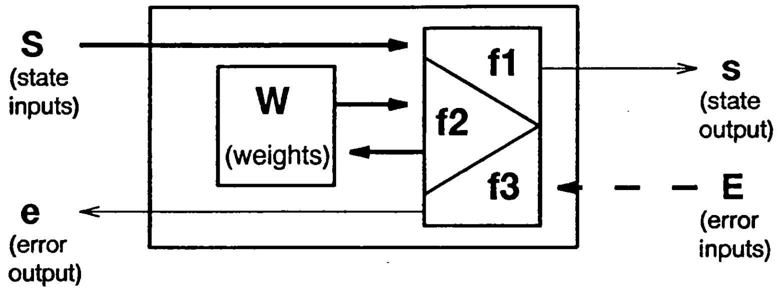


Figure 2.2. The UCL Generic Neuron [147]

Silicon Compilation is a logical extension to the evolution of special-purpose neuro-chip implementation techniques. Silicon Compilers are sophisticated programs, which process the neural network representation, and generate the VLSI definition of microchips using VLSI development packages. UCL's Silicon Compiler Project is an example of this kind of work. This project aims to develop a framework to generate the UCL Generic Neuron based architectures as target products, using Silicon Compilation [117].

2.4.2. General Purpose Neurocomputers

General Purpose Neurocomputers are programmable machines for emulating a wide spectrum of neural networks [144]. The following hardware architectures are used as general-purpose neurocomputers:

- co-processor based architectures,
- parallel processor arrays, and
- massively parallel architectures.

Co-processor Based Architectures

Conventional high performance VLSI processors are the first hardware architectures used as general-purpose neurocomputers. These processors are usually floating point, single processor accelerator co-processor boards with a local memory. These boards plug into the backplane of an IBM PC or a SUN Workstation or a Digital VAX. These co-processors speed up the execution of computationally intensive floating point multiplications in neural simulations as well as in general computing applications. They are similar to maths co-processor boards available for PCs. The performance of these products is measured by their capacity to execute the maximum size of network, and their speed of processing a network. Speed is expressed as connection updates per second (CPS). Two simple examples can be given; firstly, Intel's 80x87 series maths processor boards, developed for IBM PC compatible machines with a 80x86 central processors and secondly, Motorola MC68881 co-processor board developed for MC68020 based CPUs [142]. A number of commercial neurocomputers have been put forward using the co-processor boards accompanied by a neural network programming environment They are usually general-purpose environments with a number of neural net models and general-purpose neurocomputer hardware. Some of the examples are presented below.

The **ANZA** neurocomputer is developed and marketed by HNC [62,63]. It is designed to support any neural network model. The system comprises a co-processor boards for PC-AT, User Interface Routine Library and Basic Netware Package to support most neural network models. The ANZA boards exploit current hardware domains as execution environments. ANZA and ANZA-plus co-processor boards are based on a Motorola MC68020 plus a MC68881 floating point co-processor with 4Mbytes of dynamic RAM to store networks. ANZA is capable of implementing 30,000 PEs with 480,000 interconnections. The more recent ANZA-plus supports 1M PEs, with 1.5M interconnections and is capable of 1.5M connection updates per second during training and 6M updates during recall. The Basic Netware Package supports parameterised algorithms; the Backpropagation, the Hopfield and the Counter-Propagation.

The **TRW Mark** Neurocomputer family includes the Mark II software simulator, the Mark III parallel processor system and the Mark IV pipelined processor based system [143]. All systems share the Artificial Neural System Environment (ANSE) programming environment. Mark III comprises 15 parallel physical processors, each built from a Motorola MC68020 microprocessor and MC68881 floating point co-processor, all connected to a VME bus. Networks are distributed across the local memories of the 15 physical processors minimising the communications requirements on the common bus. Mark III supports 65,000 virtual processing elements with over 1 M trainable connections and processing 450,000 interconnections per second. Mark IV, on the other hand, supports up to 236,000 virtual PEs and 5.5 M interconnections, and is capable of processing 5 M interconnections per second.

The SAIC **SIGMA-1** neurocomputer uses DELTA [5,116,152] floating point processor board in it execution environment. DELTA FPP is a high speed floating point engine optimised to calculate an activation value for each PE. This operation involves a multiplication of two 32 bit numbers followed by an addition to another 32 bit number. DELTA is able to execute these multiplication/addition instructions at clock speeds of 11

MHz, approximating to 22 million floating point operations per seconds peak rate. The processor is based on a pipelined Harvard floating point architecture with one program memory and two data memories. The 12 Mbyte memory supports up to 3.1 M PEs, with a rate of 11 M CPS. SIGMA-1 comes together with an object-oriented programming language ANSpec and a neural net library ANSim. ANSim library is menu-driven, and contains 13 neural network models, providing an interface to dBase III and Lotus 1-2-3 software packages. The models covered by the systems include variations of Backpropagation, Adaptive Resonance Theory, Kohonen Feature Maps and the Boltzmann machine.

Most large neural network applications are executed on conventional parallel architectures [59, 144] such as **Transputers** [108]. Transputer based co-processor boards are popular as they offer a good performance for a wide range of applications, and Transputer-based systems are cheap and efficient. Similar to Transputers, other general-purpose, multi-processing RISC architectures Sparc and Mips processors also provide programmability and flexibility. Transputers are much faster than the co-processor boards for the PCs. A 20MHz T800 chip provides 10 MIPS and 1.5 MFLOPS and is 3 times faster than a 20MHz 80386 with a maths co-processor [98]. Transputers are medium-grained parallel architectures providing concurrent parallel processing with a parallel language called OCCAM [71]. T800 has 4 KBytes internal RAM and supports 32 bit external memory interface. A transputer can be attached to a maximum of four others at a time. Communications are synchronised using a simple protocol with start stop bits and message acknowledgement. The maximum data transfer between the T800s is 1.7 MBytes/sec. There are two links, and when both are used 2.4 MBytes/sec is obtained. The new Transputer INMOS T9000 has a 16 KByte cache and a 64-bit address bus [150].

Parallel Processor Arrays

General-purpose neurocomputers based on processor arrays are a result of the evolution of co-processor boards on a larger scale. Parallel processor arrays are [135] composed of a large number of primitive processing units, connected in a regular and restricted topology. An early general-purpose neurocomputer architecture based on parallel processor arrays (Figure 2.3) was proposed by Hecht-Nielsen [59]. The structure brings together N identical processors connected through an interconnection network. Each processor executes a section of the virtual network. To program the neurocomputer, the virtual PEs are partitioned across the local memories of the physical processors.

Execution of a neural network involves a continuous update of the states of the virtual PEs. Updating a virtual PE implies broadcasting the update through the network. Processors that need access to that information accept and store the update in their local system state memory. Computation, therefore is carried out through a sequence of iteration cycles. The subsequent iteration occurs in lock-step, when all the other processors have completed the previous cycle. This approach is a way of time-multiplexing [21] several PEs on each available physical processor. Performance can be increased by increasing the number of processors, or adding co-processor accelerator boards for fast execution, or high-speed memories for fast memory access.

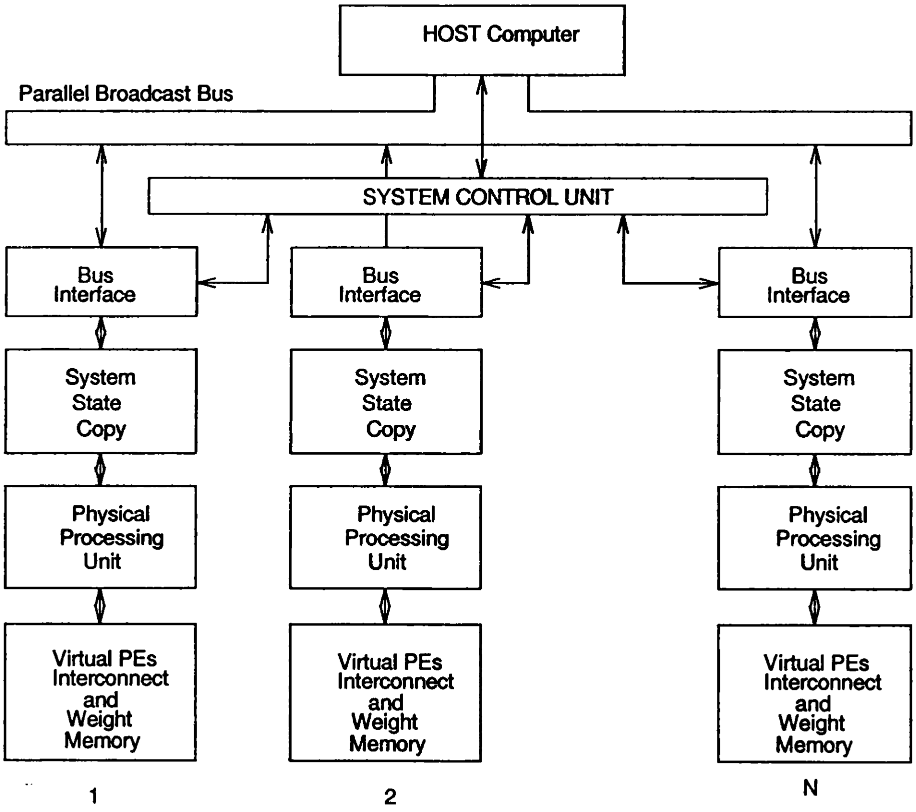


Figure 2.3. HNC General Purpose Neural Computer

Some processor arrays are built from replicable boards based on industry standard microprocessor chips such as DSPs and Transputers. Others develop custom designed neuro-microprocessors. Research groups in the USA, Japan and Europe have developed processor array based neurocomputers [143]. Here are some examples:

The **NETSIM** system is developed by Garth [50] of Texas Instruments (UK) with Cambridge University. The system consists of a collection of neural network emulation cards arranged in a 3-dimensional array structure. A PC acts as a host system controller.

Each NETSIM card is an autonomous single-board processing unit based on an industry-standard microprocessor and designed to solve NN problems. A large number of NETSIM cards can be connected via a message passing network. Each NETSIM card contains the following modules; an Intel 80188 local microprocessor, an associated program memory, a Solution engine, and a microprocessor for communications. The Solution engine is the heart of the system; operating as a back-end vector co-processor for the local microprocessor, and performing mathematical functions on the contents of the synapse/input memory. It computes the sum of products between the input vector and the synapse vector. The result is returned to the microprocessor, which in turn computes the threshold function to produce the output of the neuron, and determines the destination for subsequent transmission. The communications processor interconnects NETSIM cards to their nearest neighbours. The host, a SUN, a VAX or a PC acts as a system controller for the neurocomputer by initialising and mapping applications to the system. The system is programmable, and supports a wide range of NN models. The synapse memory size (1 to 2 MBytes) determines the number of input neurons or networks per NETSIM card. The execution time depends on the network configuration of the chosen model. A typical implementation using 125 network cards has shown a rate of 450 million CPS on the recall phase.

The **Network Emulator Processor** (NEP) by IBM is part of a complete network programming environment called COmputation Network Environment (CONE) [35,55] developed by IBM at Palo Alto. NEP is a cascadable unit designed as a co-processor for IBM PC. Up to 256 NEPs can be cascaded in a unidirectional interprocessor communications network, to support a total of 1 M virtual PEs and 4 M interconnections. A global interface to the host PC is 100 MBytes/sec inter-NEP NEPBUS service. NEP can simulate about 4 K virtual PEs and 16 K interconnections, with 30-50 total network updates per second. The length of the network update cycle can be reduced by partitioning the network across a number of NEPs [35]

Meiko In-Sun Computing Surface is a multi-processor scalable environment based on T800 transputer [7]. 96 Transputers can be embedded in a single workstation with 512 MBytes system memory and a performance of 20-400 VAX-Mips stations. An Electronic Message Link Switch enables configuration of the topology at run-time (these could be trees, grids, rings, toroids). Message paths through the switch operate on a 20 MBits/sec data transfer speed. The computing surface includes a SUN host, and up to four additional processors with dual ported shared memory in the SUN address space. There is also another proprietary VLSI chip to supervise the execution and diagnostics.

Intel Scientific Computers offers **iPSC**, a multiprocessor system which can operate concurrently with up to 128 processors as a hypercube. Each node is a board level microcomputer with 80286/80287 chip sets. Local memory is 512 K expandable to 4.5 MB. Each node contains 8 bidirectional communications channels managed by dedicated communications co-processors. The first 7 of these channels are direct links, the 8th is the global communications link to the Ethernet for program load I/O and diagnosis. The system provides a 10 Mbit/second bandwidth for internode communications. iPSC-VX is a vector concurrent system which couples vector processors to the nodes. The system can yield up to 1280 MFLOPS on 32-bit data.

The **ANNE** (Another Neural Network Emulator) [20] system has the following modules: a user interface with monitoring facilities, a high level language, Network Description Language (NDL) which is based on Scheme (a lexical scoped version of Lisp) which is used to describe NN structures; a low level generic neural network specification language called Beaverton Intermediate Form (BIF). BIF is generated by a compiler from NDL definition, then the Mapper partitions the BIF definition onto Intel's iPSC processors. During the runtime, a message passing scheme supports the communications between the different iPSCs nodes. A timing mechanism is also included to synchronise execution.

The **Giga CoNection** (GNC) [67] system contains a two dimensional array of general purpose 64-bit RISC Intel 80860 processors. Each processor has 4 MBytes of local memory and 2 FIFOs (first in first out 64-bit memory) for mesh connection. The 80860 contains a floating point pipeline providing high performance for multiply/add operations. Hiraiwa et al. report 1 Giga CPS on a 128 processor system in the simulation of the Backpropagation model. GCN-128 system provides two pipelined communications paths. The Sony group is incorporating the system as a super chip.

Ariel [46] is a multiprocessor architecture based on coarse-grained processing modules. The modules make use of fast digital signal processors and large semiconductor memories to provide the necessary throughput and storage capacity. Each Ariel module consists of a 32-bit fixed/floating point DSP (TMS320C320), a 32-bit general-purpose processor (GPP), 128 MBytes memory and several high speed communications ports, and a dedicated disk memory unit. The Ariel architecture uses bus communications to achieve generality over a wide range of neural network models.

Sandy/8 is another digital neurocomputer architecture [85]. Sandy/8, a research system has been developed at Fujitsu Laboratories in Japan. It consists of 256

TMS320C320 floating point Digital Signal Processors, each with 2K internal and 64K high speed external RAMs. A ring communications topology with 67 MBytes/sec band width serves parallel processors. The 256 processor system can reach 587 MCPS during a Backpropagation training, involving networks with more than 256 neurons. Sandy system is also suitable for conventional image processing and vector processing as well as neural network applications.

Warp is a MIMD one-dimensional systolic array computer designed at Carnegie Mellon University [103, 120]. It consists of 10 identical cells in a linear array with a peak performance of 100 MFLOPS. Each cell contains one ALU and a multiplier, with a 4KByte of 152-bit word micro-store and a 4KByte of 32-bit RAM, and can deliver 5 MFLOPS. Cells can be programmed separately for different operations, and data can be pipelined through the cells using the two data channels per cell. The Warp machine is designed to interface with VAX 11/780 with a 1 MByte of memory and 24 Mbyte/sec bandwidth. High level routines are carried out on the host computer.

CNAPS (Connected Network of Adaptive Processors) [8] is a general purpose neurocomputer chip with 64 processors, each containing 4 KBytes of local memory. It is a SIMD machine based on a linear array of digital signal processor-like nodes (PNs). CNAPS Server has 256 PN processors and provides an Ethernet LAN interface to connect to a SUN workstation [54]. The PN processors are designed for traditional neural network applications. Each PN has an associated 4KBytes internal memory, making altogether 1 MByte per system. Also an external data storage facility provides 8, 16, 32, 64 MBytes global memory for the array of PNs. The processor array is connected by three buses; OUT bus, PNCMD (PN CoMmanD bus) and an IN bus. A single controller sequences the linear array of PNs. The sequencer places data onto the IN bus and forwards sequencing commands to the PN array. The PNs execute the same instruction at each clock cycle. More than one PE can be mapped onto a single PN. Also time multiplexing can be used to assign multiple PEs to a PN or divide a complex PE onto a number of PNs. The **CodeNet** [8], provided by Adaptive Systems, is a high performance commercial programming environment to support its CNAPS neurocomputer. The complete system consists of a server, a programming environment (CodeNet) and the CNAPS array processor. CodeNet consists of CNAPS Programming Language (CPL); Applications Programming Interface (API) and a library enabling users to include CPL programs in *C* code. CPL is a modular programming language also providing low level access to CNAPS hardware. The environment also contains a set of tools to debug, modify, and execute CPL programs. Novel applications can be developed by using the library calls. An 8 chip configuration can update 2.3 billion CPS in learning and 9.6 billion CPS in recall.

Massively Parallel Arrays

The **Connection Machine** is a fine-grained massively parallel machine. Its recent version CM-2 is a data parallel computing system made of 65535 processors. It is an SIMD architecture, which requires a sequencer to break down the high level representation into low level processing and memory operations [3, 64, 155]. 64K processors can take orthogonal topologies and grids with arbitrary dimensions are supported. It can be configured as a 2-Dimensional array of virtual processors. Each processor is a bit-serial processor with 4K memory (32 MByte for the computer). Interprocessor communications are handled by a message passing system. The CM has two communications: each processor is linked to the nearest neighbour and 16 others in a n-cube geometry whose binary addresses are 1 bit different. The system supports parallel versions of *C* and *LISP*. It is an expensive machine, and complex to program with many I/O controllers, sequencers, and interfaces. Neural network simulations on CM-2 can reach up to 40 MCPS in Backpropagation training, and 180 MCPS in recall [155].

AAP-2 is another massively parallel SIMD processor array which was used for neural network execution [149]. It consists of 65,536 one-bit processors with 8 Kbits of local memory. The processors are configured as a 2D (256x256) array with high-speed data transfer mechanisms. The language for AAP-2 is a parallel programming language; AAPL which is an array-oriented language consisting scalar and array operations. The performance of the system can reach up to 18 MCPS in Backpropagation training.

Conclusion

Special-purpose neurocomputers are often *neuron-based* architectures aiming at a massively parallel execution of certain models and applications. General-purpose neurocomputers, on the other hand, are general, flexible, cost-effective and scalable. Early general-purpose neurocomputers are co-processor based architectures aiming to speed-up demanding floating point operations involved in neural network simulations. Later, processor array based architectures focused on the efficient execution vector/matrix operations as *vector-based* machines. Massively parallel general-purpose architectures can potentially be used for mapping neurons or vector/matrix operations. On these architectures the complexity is pushed onto software for the efficient exploitation of hardware.

2.5. Summary

This chapter provided a review of neural computing with its rapidly increasing number of models, applications, programming and execution environments.

Programming environments are a medium for mapping or transforming real-world problems onto computer programs and also for mapping computer representations onto hardware. It is felt that, within the programming environments, algorithm libraries are particularly useful as they provide the most flexible, user-friendly means for neural network programming. In chapter 5, an analysis of neural network representations is used to argue for a matrix-based library which is capable of capturing common neural network operations, also facilitating the mapping and execution processes on parallel hardware.

Neural network execution environments were studied in two categories; special-purpose and general-purpose neurocomputers. The main difference is in the performance and flexibility of these systems. Special-purpose neurocomputers are favoured for processor simplicity, and thus cheaper production rates and high performance. But in some cases, the architecture of special-purpose neurocomputers reaches a level of complexity comparable to the conventional systems. General-purpose neurocomputers aim to serve as fast execution platforms for a wide range of neural network models. Most general-purpose architectures can also be used as accelerators for other computationally demanding problems.

In terms of mapping strategies, two approaches have been noticed: *(i)* structural mapping - particularly, special-purpose neurocomputers emulate neurons on parallel physical processors, and *(ii)* computational mapping - conventional computers and general-purpose neurocomputers optimise the computations which are the core of the neural network simulations on co-processor and accelerator boards. A generic representation and mapping strategy is required to exploit current general-purpose parallel architectures in a cost-effective framework.

Chapter 3

Pygmalion and Galatea Projects

This chapter presents and assesses two major European neural computing projects; Pygmalion and its successor Galatea. Although the two projects share many common features, they differ in their intermediate-level representations and target hardware architectures. The Galatea General Purpose Neural Computer is presented in detail, as it is the test and implementation domain for mapping strategies developed in this thesis.

3.1. Introduction

Pygmalion and Galatea are two major European neurocomputing projects. The author of this thesis has been involved in both projects, and has contributed to the representation techniques developed in both projects. This thesis aims to establish a generic representation technique for neural networks. An assessment is made on the two representation techniques adopted by the Pygmalion and Galatea in chapter 5. Furthermore, the Galatea GPNC is used as the implementation domain for the mapping strategies developed as part of this thesis work. Chapter 7 presents the mapping efforts on the Galatea GPNC simulator. For these reasons, it was felt that these two projects should be presented separately from the rest of the Neural Computing survey.

3.2. Pygmalion Programming Environment

The Esprit II Pygmalion project [14, 15, 18, 146] aimed to promote the application of neural networks by European industry, and to develop standard computational tools for the programming and simulation of neural networks. Pygmalion programming environment is based on a generic hierarchical data structure, covering the most popular neural network models. The Pygmalion data structure, *system* contains a tree of *networks, layers, clusters, neurons* and finally *synapses*. The Pygmalion environment comprises the following modules: (Figure 3.1)

- **The Graphics Monitor** allows users to execute and monitor neural network simulations. It uses the X Windows graphical interface protocol on SUN workstations. A menu-driven system permits the user to choose an application, initialise the execution, open and close graphical windows and monitor the

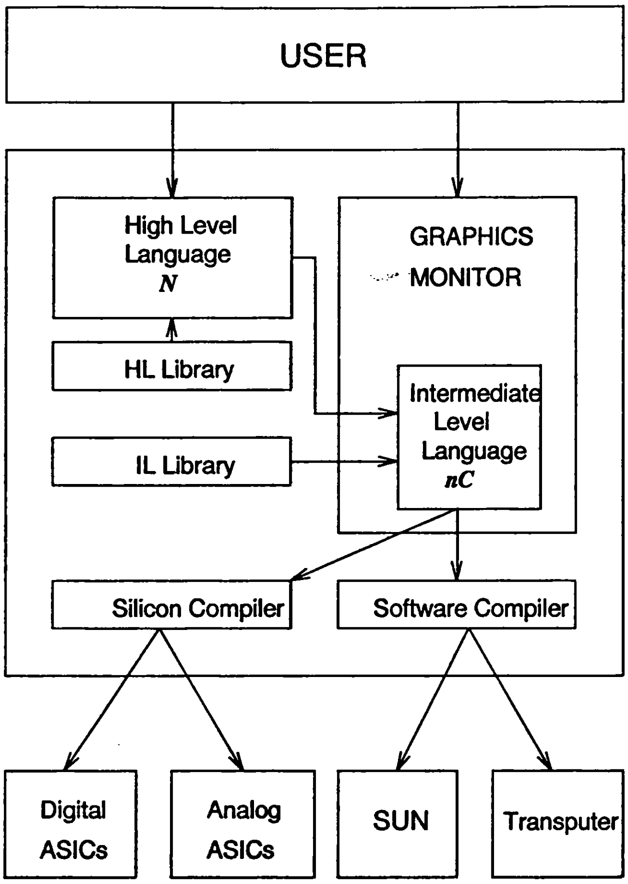


Figure 3.1. The Pygmalion General Purpose Programming Environment

execution. Trained or partially trained networks can be saved or loaded. Two types of windows are available; Top Windows, providing facilities for controlling the simulation, and Level Windows allowing the display of the network status for each level in the system data structure.

- **The High-Level Language** *N* provides a user-friendly definition of neural network algorithm and applications. *N* is an Object Oriented language, based on *C*++.

- **The Algorithm Library** module contains a collection of the most widely used neural models written in the high-level language *N*. A number of rules guide users who want to develop new applications. Users can interrupt and store applications and continue to execute these later.

- **The Intermediate-Level Language** ***nC*** is a machine-independent low-level neural network specification language. *nC* is a subset of *C*, consisting of a massive hierarchical data structure which allows the representation of all possible objects in most neural models. The *nC* data structure tree contains rules, parameters and substructures repeating the same pattern until the synapse level. The Backpropagation, the Self-Organising Map, the Hopfield and Competitive Learning models are provided in this language as parameterised routines. A translator automatically generates *nC* representation from the high-level language *N*. All run-time user requests and debugging operations are carried out on this intermediate-level representation.

- **The Compiler/Translator** allows the porting of the neural network representations for execution onto different machines. Pygmalion software is developed in *C* on Unix SUN3 and SUN4 workstations. To improve execution speed a series of compilers has been planned to enable the porting of the final *nC* representation to different target architectures, and their execution on these machines. A compiler has been developed which translates *nC* to parallel *C*, and this representation is executed on Transputer based machines.

The Pygmalion programming system is a fully integrated software environment. The code generation cycle involves the High Level Language (HLL) definition of neural network applications, the translation to Intermediate Level Language (ILL), and the porting onto a number of different hardware platforms. The graphical interface provides a fast user-friendly manipulation on SUN workstations.

Neural Network Representation and Parallelism in Pygmalion

The Pygmalion programming environment adopted a two-level representation strategy. An object-oriented, high-level, neural network definition language *N* is used as user interface [104], and a machine-independent, intermediate-level language *nC* is used for low level representation [18, 147]. *nC* is a subset of *C*, based on a hierarchical data structure called *system* (Appendix A.1.). The *nC system* data structure is a chain of pointers, and pointers to pointers, finalised by data or functions at the lowest level of the data structure; a neuron or a synapse. A number of algorithms are provided in *nC* in a parameterised format. These can be used executed using menu-driven facilities of the Graphics Monitor.

Explicit parallelism is accommodated in *nC* with the control statement PAR. Any rule or loop statement preceded by the PAR statement can be potentially executed in parallel. Attempts have been made to map *nC* onto Transputers [19] and two projects focused on the automatic generation of the UCL Generic Neuron on silicon VLSI chips from *nC* definitions [117, 147]. The *nC* representation is further assessed in chapter 5, in the context of different representation techniques.

3.3. Galatea Neurocomputing Project

The Esprit II Galatea project is developing an integrated software and hardware system for the development and execution of neural network applications. The Galatea Project comprises the following modules;

1 - A **General Purpose Neural Computer** (GPNC) hardware, with efficient support for a wide range of neural networks.

2 - A **Neural Network Programming System** (NNPS), a sophisticated neural programming environment, allowing the efficient use of the GPNC, domain-specific processors, conventional parallel computers and workstations.

3 - A **Silicon Compiler** for rapid and low cost Application Specific Integrated Circuits (ASIC)

A number of applications exploit this high performance and general neural computing platform. A neural network based industrial vision workpackage investigates the potential use of neural technologies in industrial applications to improve Surface Mounted Device assembly technology. Similarly, the industrial vision workpackage has been applied to video-grading of damaged oranges. Both applications form a testbed for the Galatea GPNC. Optical Character Recognition (OCR) is seen as an ideal application for the Neural ASICs, which will be automatically generated by the Galatea Silicon Compiler. The aim of this package is to scan a text bitmap and convert it to ASCII form involving implementation of neural models suitable for character recognition and mapping onto silicon. A commercial prototype PC board is to be produced which will provide high performance and speed in optical character recognition tasks.

3.3.1. Galatea General Purpose Neural Computer

Galatea GPNC is a heterogeneous distributed architecture which brings together generic modules, called Virtual Machines (VM) (Figure 3.2). The building block of the Galatea GPNC, a VM, contains a Communications Unit and an Execution Unit. The Communications Unit is responsible for coordination of communications with the host and other VMs. An intermediate-level matrix-based language called *VML* [140] is the common language between VMs and the host. All VMs communicate, interpret and execute *VML* concepts. An interpreter for *VML* has been developed at UCL to allow the simulation of the GPNC.

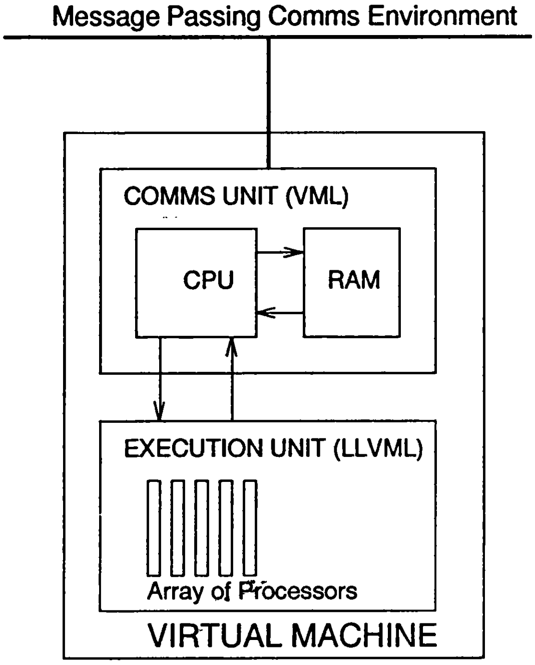


Figure 3.2. The Virtual Machine

The *Communications Unit* links VM to a bus which is connected to a SUN workstation host. The CU contains a Central Processor and local memory, and is able to store and interpret *VML* code, carry out scalar arithmetic operations and fire low-level instructions to the Execution Unit. These low-level instructions are based on vector and matrix arithmetic, and a low-level machine specific representation called Low-Level *VML* (*LLVML*).

The *Execution Unit* of VMs is composed of fast matrix multiplier general-purpose neurocomputer boards. Two boards are being developed in parallel in Siemens (Munich) and Philips (Paris). Although both boards follow the same general-purpose philosophy, they differ in local memory management, execution speed and data representation. For efficient execution on these boards, *LLVML* has to meet the requirements of correct data placement and typing. Hardware-specific features, such as the ability to multiply four vectors simultaneously - as is the case for the Siemens board - must also be exploited at this level.

A number of VMs could be plugged to a SUN workstation enhancing the power of the GPNC. The common language between these VMs is *VML*, and the communications medium which carries *VML* instructions is a message passing scheme. Code generation for this multi-VM coarse-grained parallel architecture is the task for the Galatea Mapper. The run-time operations are monitored by another process, the Scheduler which takes over after the initial mapping is completed.

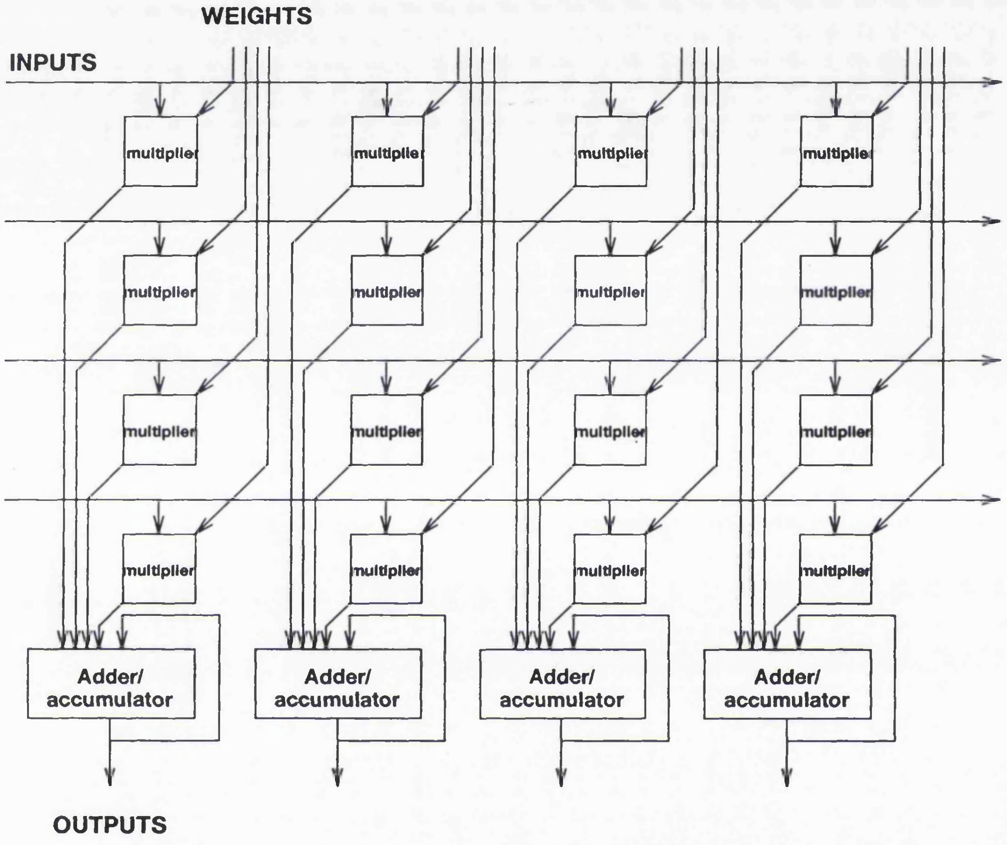


Figure 3.3. The Siemens Architecture

The Siemens Architecture

A series of architectures is being developed by U. Ramacher and his group at Siemens to emulate neural network models on hardware [122-124]. The design of their architecture focuses around cascadable modules of matrix multipliers. Each multiplier represents a synapse, where the multiplication of the weight value by the neuron's state takes place. Each column corresponds to one neuron unit. Figure 3.3 shows the conceptual structure of a 4-neuron module with 16 synapses. This module is composed of 4 columns, each comprising 4 multipliers. Every multiplier receives in parallel, a weight and an input value. At the bottom of each column there is an adder that sums the 4 weighted inputs calculated by the multipliers, and an accumulator, which stores the partial weighted sum.

The Siemens architecture offers good performance by using the matrix of multipliers and fast memories. The parallelism is achieved as a result of the simultaneous processing of multiple patterns. The system architecture is flexible and general enough to implement a wide range of models. But the full performance can only be achieved in cases where the networks are fully connected and multiple pattern processing is allowed.

The Philips Architecture

The Philips architecture which is developed in Philips Labs in Paris, is a neuro-chip efficient in matrix multiplication and addition operations. It is a fully digital CMOS VLSI chip that allows various kinds of network models to be executed [40, 107, 141, 141]. Figure 3.4 shows the conceptual model of this architecture. The model contains a synapse memory implemented in RAM. This matrix memory allows the storage of *NxN* weight values coded in binary 8 or 16 bits. The neural state register contains *N* state values which are obtained from the multiplication of the *N* inputs by the synapse matrix. Both multiplication and addition processes are executed in parallel, but each neuron is treated serially in the state update. The threshold function is evaluated off-chip, either by a dedicated hardware, or by a standard processor. After this evaluation, the states are stored back in the Neural State Register.

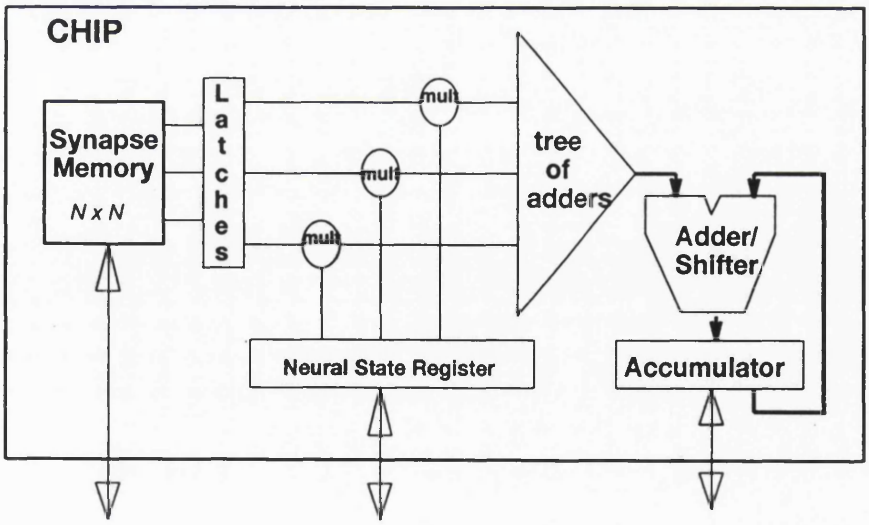


Figure 3.4. The Philips Architecture

The first version of this architecture was implemented using 1.6 micron CMOS technology, containing an 8-bit 64x64 synaptic memory. This version did not include the learning routine. The Philips architecture overcomes the necessity to have fast memories by integrating the synaptic memory on-chip. This localised RAM approach is favoured as it reduces the number of parallel feeds to the weight multipliers.

3.3.2. Galatea Programming System

The Galatea Neural Network Programming System is based on the same principles as those outlined in the Pygmalion system. However, it is more sophisticated, and as a design philosophy, a parallel distributed processing approach has been adopted. It encompasses the following modules:

- **The User Interface** is a sophisticated tool consisting of a set of independent programs. It has been built using the Motif graphics application builder. Motif uses the X Windows graphics protocol with a set of customised library functions. The user interface has three modules; a Graphic Monitor, an Execution Monitor, and a Debugging Monitor. The Graphic Monitor is responsible for opening various display windows for the input and output patterns, system error graphs, bar charts etc. The Execution Monitor is used to initialise, interrupt, save or load the execution. The Debugging Monitor allows line by line tracing of the intermediate level code during execution. The three programs are independent, and they communicate with the system through the central Scheduler.

- **The High-Level Language** *N* is an Object Oriented language. It is a further development on *N*, from the Pygmalion programming environment. Again, it is supported by a High-Level Language Library containing the code for the most popular neural models. Users can use the HLL library, or write their applications in *N* which is similar to *C++*, or use the Systems Application Builder (SAB), which allows the development of applications using graphic tools.

- **The Intermediate-Level Language** *VML* is a vector-matrix based language. Vectors are considered as one-dimensional matrices. The instruction set for *VML* contains matrix and scalar arithmetic operations, data transfer and control commands, and file I/O statements.

- **The Mapper** is responsible for the efficient partitioning and distribution of the *VML* neural network representation. It schedules the operation, and downloads the code to VMs generating appropriate data exchange instructions.

- **The Scheduler** is a run-time process handling user requests and communications between the VMs and the host. It runs like a Unix deamon, passively waiting for requests from users or the modules of the system.

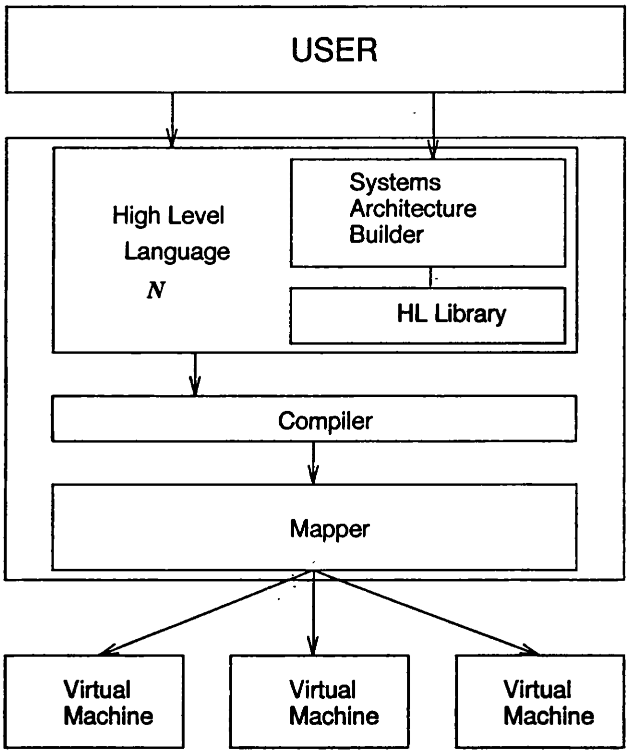


Figure 3.5. Galatea Programming System

The code generation process (Figure 3.5) for the Galatea NNPS is a complex process. Users defining applications in the high-level language *N*, prompt the *N* to *VML* compiler to generate vectorised *VML* code. The compilation process also generates a Symbol Table that contains a correspondence between *N* and *VML* concepts. Then, the Mapper processes the *VML* representation, and partitions it to exploit the parallel execution environment. The Mapper generates parallel *VML* code with appropriate data transfer instructions. This code is downloaded to the VMs, the GPNC is prepared for execution and awaits the start signal from the user. Users can initialise execution using the Execution Monitor, and the Scheduler issues the start signal to the VMs so that the parallel execution starts.

Neural Network Representation and Parallelism in Galatea

The Galatea project has adopted a two-stage representation scheme, involving an object-oriented, high-level language and a matrix-based, intermediate-level language. The high-level language *N* meets the user requirements of design simplicity, flexibility and modularity. The intermediate-level language *VML* aims to exploit vector-based general-purpose hardware boards efficiently.

VML consists of a set of scalar and matrix operations with a *C*-like syntax. It is an interpreted language containing; control statements, I/O operations and arithmetic instructions. The *VML* interpreter which has been developed at UCL, is used to execute *VML* programs on SUN workstations. The interpreter creates an executable image of the code, which can be either executed or used to generate Low-Level *VML* instructions. *LLVML* will be executed on high-performance matrix operator VMs, currently being manufactured at Siemens and Philips.

Two levels of parallelism are possible on the Galatea GPNC. On the higher level, neural network applications can be distributed across a number of VMs. Further parallelism can be exploited within each VM by mapping matrix based operations onto processor array based Execution Units. This thesis work involves the high level partitioning and mapping of matrix based neural network representations onto a number of VMs.

3.4. Summary

This chapter provided a review of the Pygmalion programming environment and the Galatea Neurocomputing project with a special emphasis on the Galatea GPNC and its programming system.

The Pygmalion programming environment has all the features common to general programming environments; an object-oriented, high-level language, an intermediate-level representation, an algorithm library and a set of compilers for a number of target hardwares. Pygmalion principally targeted *neuron-based* hardware architectures suitable for silicon compilation and similar to the UCL Generic Neuron presented in chapter 2. Pygmalion partly achieved its aims of increasing interest in neural computing in Europe, providing the end-user with a set of algorithms and an experimental tool. However, it has not become a standard for neural network software environments nor has it provided a high performance execution.

The Galatea GPNC marks a change in direction from a *neuron-based* to a *vector-based* philosophy. It comprises high performance Virtual Machines that exploit vector operator Execution Units. The vector-based trend is also reflected in the intermediate-level language, *VML*. Although Pygmalion and Galatea share the same high-level language *N*, their intermediate-level languages are remarkably different. This is further investigated in chapter 5, in the analysis of neural network representations. The development of the GPNC is still in progress, and the Galatea GPNC is the test domain for the mapping strategies developed in this thesis. The mapping efforts for the Galatea GPNC are presented in chapter 7.

Chapter 4

Analysis of Neural Networks

In this chapter, three neural network models, the Hopfield, the Self-Organising Map and the Backpropagation are analysed in three case studies with appropriate real-world applications. The analysis focuses on the computational requirements and possible structural mappings for these models. A comparison of the three models is presented, and the potential benefits of modular, multiple-neural-network architectures are discussed.

4.1. Introduction

Neural networks models appear in a variety of topologies with a number of training and recall procedures. Chapter 2 presented a survey, which covered a number of models in terms of their main characteristics and popular applications. In this chapter, using three case studies, three neural network models are analysed. In these case studies, the Hopfield nets, the Self-Organising Map and the Backpropagation model are chosen as they together contain properties common to most neural network models. Each case study presents the model, the application and the results, and a computational analysis is followed by structural partitioning and mapping examples. Finally a comparison of the three models is presented, and the strengths of the multiple-network models are discussed.

4.2. The Hopfield Networks

The Hopfield networks are prime examples of recurrent nets. They are called recurrent because outputs of the neurons typically affect the inputs of the same neurons. Because of this positive feedback, the main problem with recurrent nets is to achieve stability, as the outputs of the neurons may never converge to stable states, but change their states continuously in a chaotic fashion. In 1982, J. Hopfield proposed a network of binary processing elements (on-off devices) [72] and proved that it converges when a number of conditions are satisfied. Since then, a huge amount of research has been carried out on applications, hardware implementations, the pattern storage capacity of the net and the possible learning algorithms.

4.2.1. The Model

A Hopfield net consists of a number of fully-connected processing elements (Figure 4.1), with all neurons connected to the others with symmetrical weight values. To achieve convergence, the connection weights matrix (W) must obey the following rules: $W_{ii} = 0$ and $W_{ij} = W_{ji}$. Binary Hopfield neurons are simple processing elements performing the following tasks: each neuron independently sums its inputs, thresholds the sum and outputs one of the two states, either 1 or -1. Initially, the states of some neurons are externally clamped values, later the neurons take values using the outputs generated by the other neurons. At the beginning, neurons change their states frequently, later the rate of change decreases, and finally the network stays totally stable. The states of the neurons at this stage represent the response of the network to the initially clamped stimuli. The overall state of the network is described as an energy function which continuously decreases until the network reaches a stable low energy state.

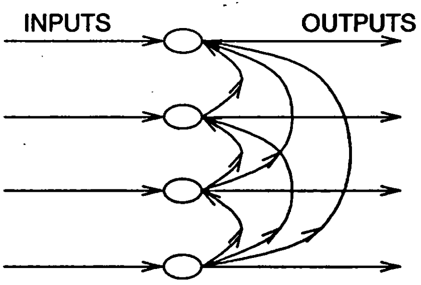


Figure 4.1. The Hopfield Net

Hopfield claimed that the net could be used as a Content Addressable Memory (CAM), storing approximately up to *0.15N* number of separate memories with *N* number of neurons. These memories must be distinct enough not to create spurious states in recall or merge two or more patterns as a result. An approximate condition for accurate recall is that each pattern must be *0.5N* Hamming distance units apart from the other patterns in the dataset - Hamming distance being the count of mismatches between the elements of two binary vectors. If accurate results are required in recall, an optimum set of weights must be found. Hopfield suggested the use of the sum of the outer products of all the input vectors in the calculation of the *W* weight matrix. This is a primitive learning procedure which is used to form an orthogonal weight connection matrix. A Hebbian-like incremental learning procedure was also suggested which results in pseudo-orthogonal weight matrices. Research continues to develop algorithms that store the highest number of patterns on a net with a given number of neurons [13, 87].

Hopfield claimed that his net is biologically plausible because neurons can update their states asynchronously, independently of each other. Subsequent research has shown that both synchronous and asynchronous state update routines result in convergence. Hopfield also showed the net can be used with continuous valued inputs by changing the hard limiting threshold function to a nonlinear sigmoidal function [73].

The most powerful applications for the Hopfield net are optimisation problems. Hopfield and Tank made two demonstrations; the Travelling Salesman Problem [74] and an Analog/Digital converter. Both cases involve considerable effort to set up the connection weight matrix. Once this matrix is set up, the net is capable of producing an adequate solution to the optimisation problem. In optimisation problems the constructed weight space resembles an energy surface with many hills and valleys, with the deepest valley being the global optimum solution. Unfortunately, Hopfield nets tend to find local minima, rather than reaching the global minimum. To overcome this difficulty a procedure based on the process of metal annealing is added to the algorithm. This is a stochastic state update procedure that suggests a start with a high temperature to make sure the network explores the global search space, the temperature is gradually reduced, and the network settles to the global minimum. But this routine is computationally too demanding to assure a global solution within a practical period.

4.2.2. Case Study: Pattern Recognition

The auto-associative nature of the Hopfield nets can be exploited in pattern recognition tasks as the network tolerates a high degree of noise and can operate with partial or incomplete information. This is extremely desirable in pattern recognition problems, and a real-world example is presented below:

A telecommunications system involves the transmission of binary patterns between two satellite stations in the atmosphere, which is prone to random noise. The patterns transmitted from one station reach the other, in a partly corrupted state depending on the noise level. The Hopfield net can be used to reconstruct the noisy or incomplete input patterns as a Content Addressable Memory. In this case, a dataset containing 12 orthogonal patterns each with 8x8 grids of elements (Figure 4.2) has been used. The two-dimensional nature of the patterns eases the selection of orthogonal patterns, as patterns can be designed visually. The weight matrix is set up by taking the outer product of all the pattern vectors and adding them up. Random noise is added to the patterns by randomly switching on/off the elements of pattern grids.

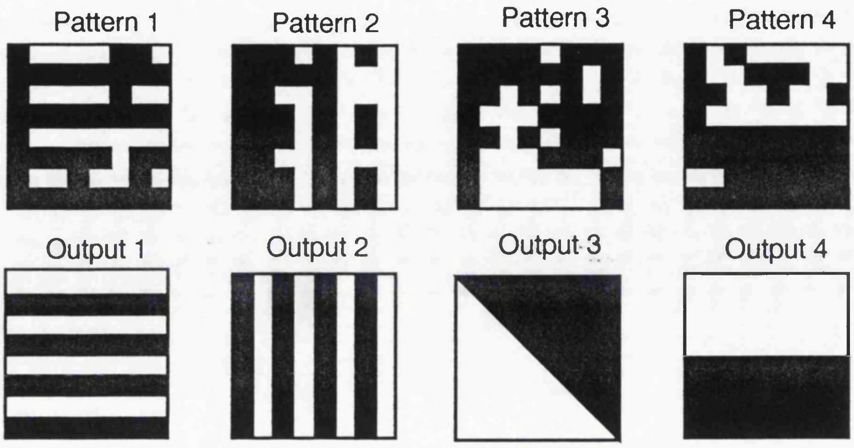


Figure 4.2. The Patterns for the Hopfield Net

In the recall phase, a series of synchronous state update operations takes place, until the network settles. This is done by updating states of all neurons simultaneously (in a lockstep mode), using the outputs of other neurons from the previous step.

A Hopfield net simulator has been coded in *C* to solve this pattern recognition problem, and to carry out a computational analysis. The simulation of the net reaches convergence rapidly, finding one of the original pattern vectors as an output. Increased noise level, or too little information about the input patterns, causes spurious states to emerge in the outputs. Even patterns with 50 % of the original data points result in complete patterns with a fast convergence of approximately 4, 5 iterations.

4.2.3. Computational Analysis

This size of Hopfield net simulation (64 neurons) on a SUN Sparc workstation converges in a short time. Profiling the execution by using the standard Unix facility 'gproff' produced the following results: for all 12 patterns the net converged to correct patterns in under 4.7 seconds. Other computational operations in the simulation, such as printing out the outputs and reporting the states of the neurons also consume a remarkable amount of computational time. Although the net provides an answer in a short time, a real-time usage would necessitate a much faster execution rate. This can only be achieved on high performance parallel hardware, or VLSI hardware implementations of the network.

The learning procedure for the Hopfield net involves setting up the weight matrix using a set of patterns which are orthogonal one against another. This is done once, and it does not pose a big computational load. The recall procedure is the computationally intensive part of the Hopfield net simulation, and it involves an iterative procedure of state update. This procedure consists of a series of multiplications of inputs by weight values, followed by a summation. The sum is then applied to the threshold function which can be a simple decision mechanism associated to a value (such as 0) or it may be a nonlinear squashing function with bounds -1 to 1. The first is the case for a binary Hopfield net, and the second one is used on the continuous valued version. In the case of using a hard-shoulder activation function, computational requirements are minimal, in fact, in *C*, this function is an *if* control statement. But the continuous valued Hopfield requires a sigmoidal or tangent hyperbolical activation function. The tangent hyperbolical function can be called by using the *C* library routine *tanh()*, or alternatively, a look-up table can be set up, and this table can be used. The use of the system built-in functions is computationally more demanding, as their implementation involves a number of multiplications carried out in double precision arithmetic. The option of using look-up tables requires an understanding of the activation function's characteristics. The resolution of the table must be organised properly, where the rate of change is high, more data points on the table are necessary.

A typical state update operation for a single neuron involves the following operations:

$$S_i = I_i + \sum_j O_j.W_{ij}$$

$$O_i = \frac{1}{1 + e^{-S_i}}$$

The most expensive single operation here is the exponential function, although it is carried out only once for each state update. Most of the computation is focused on the *dot-product* operation which involves the element by element multiplication of the pattern vector by the weight matrix. For N neurons this operation can be executed simultaneously. A total of N^2 multiplications for each state update step can be done in parallel. Each multiplication is followed by N additions, N thresholding operations and finally, the generation of the outputs for the net. A higher level process checks the convergence by comparing the current state of the net with the previous state, and decides whether the convergence has been achieved.

4.2.4. Parallel Hardware Mapping

Two cases can be considered for the structural mapping of the Hopfield net onto parallel hardware.

1 - Fine-grained parallel mapping - In this case each neuron is mapped onto a single physical processor, which can be extremely simple. The processors carry out the following tasks: they sum their inputs, threshold the sum and transmit the result to the other neurons. An higher level central process checks all neurons' outputs and reports when all neurons stop changing states.

To test the feasibility and viability of such a system the Hopfield net has also been simulated on a number of parallel processors. Unix TCP/IP Sockets are used as the communications medium between the independent processors. Later, this feasibility study involving TCP/IP Sockets is used in the implementation of parallel *MATLIB* library and Galatea GPNC simulations. A 4 neuron Hopfield net simulation, involving 5 Unix processes, has been implemented. The first process is a server which expects 4 slave processes to plug in. When these 4 identical processes are initiated the execution starts. The processes communicate by explicit blocking data exchange statements. Every write request to the server is matched by a read request at the client end and vice versa. This type of scheduling is an example of intertwined parallel processing, and it is synchronised from the beginning of the execution.

Synchronised or independent, the communicating processes can potentially create a massive message traffic as the number of neurons are increased in the system. Assuming a bus-based architecture, the bandwidth requirements increase polynomially in proportion with N^2 when the number of neurons increase linearly. Moreover, to implement N^2 number of connections as separate communications channels on hardware is not practical because of the massively increasing wiring requirements. The following procedure has been suggested and implemented to reduce the interprocessor communications [138]:

"Only neurons which change their states transmit their outputs to the others." This method assumes a local storage facility which enables the storing of the previous state to compare with the current state. Increasing the local memory use, for the reduction of the interprocessor communications is a typical trade-off, frequently used in the manufacturing of computer hardware. But this method increases the cost of unit production of the processors.

2 - **Coarse-grained structural mapping** - this method involves using fewer, but more powerful processors with larger local memory (Figure 4.3). In this case, a group of neurons is mapped onto each processor, reducing the communications requirements. Each processor emulates a group of neurons by time-multiplexing the operation.

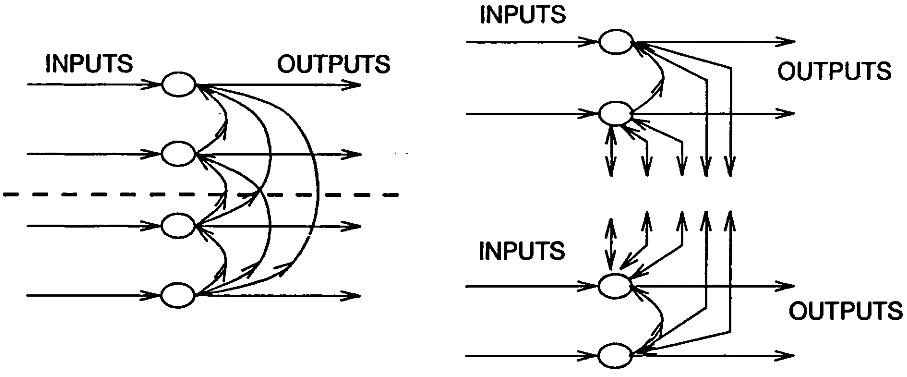


Figure 4.3. Structural Mapping of the Hopfield Model

Both types of mapping presented are based on the neuron-based assumption for the algorithm. As an alternative view, the Hopfield model can be seen as a vector mapping process involving a series of vector by matrix multiplications. This interpretation of the Hopfield net is advocated in chapter 5, using a clear and compact matrix-based representations of the algorithm.

4.3. Self-Organising Maps

The SOM has been originally inspired by the discovery of various topological feature maps in the brain [91]. These maps include retinotopic maps and orientation sensitive maps in the visual cortex, and tonotopic maps in the auditory cortex. Neurons in these parts of the brain react to a specific type of stimuli, ignoring the other types, and cluster the input vector set in a self-organised manner. The neurons themselves are grouped together with increasing degrees of sensitivity to specific type of stimuli; the sensitivity is maximised in the centre of each neural cluster. To mimic this kind of neural behaviour, Kohonen originally put forward the self-organising feature maps [88]. He demonstrated that an optimal mapping of a multi dimensional vector space can be constructed and used in pattern recognition tasks with a high degree of accuracy [126].

4.3.1. The Model

The SOM is a two layered network (Figure 4.4). The first layer, or input layer, contains *n* neurons, where *n* is the dimensionality of the input vector set. The input patterns are clamped onto this layer, so the neurons in this layer capture the activation levels on the respective dimension. The second layer which is the output layer, is usually organised as a two dimensional grid. The nodes in the output layer can be considered as fully interconnected, and the physical neighbourhood relationship and distance form the basis for the interconnection strength in this layer. All output nodes are also connected to all the input neurons with weight values which together constitute vectors of the same number of dimensions with the input vectors. Output nodes together form a vector field, and each node is a single vector in the vector space.

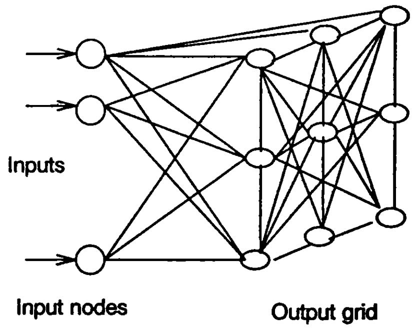


Figure 4.4. The Self-Organising Map

The SOM training [99] algorithm is as follows;

1 - Select a suitable output grid, initialise weights with small random values, define a large neighbourhood area with a neighbourhood distance, set a small gain coefficient (or the learning rate) for the weight update (typically 0.1 - 0.5), and normalise the input vectors (it can be done in two ways [37] and both methods work successfully).

2 - Clamp an input vector to the input neurons, calculate the Euclidean distance of the input vector to the output vectors, establish the closest output vector, and assign that node as the winner. Various measures can be used in the distance calculation, and the Euclidean is the most popular one.

3 - Within the current physical neighbourhood distance of the winner, update the weights, so that all output vectors get closer to the input vector.

4 - Repeat steps 2 and 3 for all input vectors.

5 - Narrow the neighbourhood distance and decrease the gain, and repeat 2, 3 and 4 until the neighbourhood is decreased to one node.

As a result of these steps, each output node specialises in a subset of the input set, and the neighbouring output nodes represent close vectors in terms of the Euclidean distance. The distribution density of the input vectors is reflected in the distribution density of the output vectors.

Once the weights are trained, a novel vector can be presented to the network and by finding the best matching output node, the class identity of this vector can be established. This forms the recall operation for the SOM.

A number of parameters influence the performance and the accuracy of the results of the SOM. These are the neighbourhood and the gain settings, and the number of nodes in the output layer. Optimum settings for these parameters is matter for research, and currently most applications involve a series of trial and error runs.

4.3.2. Case Study: Clustering Neural Spikes

In this case study, the SOM is used in clustering the neural spikes from the rat's hippocampus as a novel technique in the Anatomy field. Datasets consist of electrical activity readings of neural spikes recorded from outside hippocampal neuron cells. This method of extracellular recording, encounters the difficulty of confidently isolating the activity of the single neuron from surrounding neurons [82]. Several techniques have been developed to examine the neural electrical activity, or a spike, to identify the origins. These examinations involve the classification of the neural spike amplitude and shape [132], and most of these classification techniques require representative samples of the classes within the dataset. An unsupervised classifier, or a clustering algorithm is more advantageous in this case, since no information about the cluster centres is available in advance.

Traditional clustering methods can be traced back to the K-means nearest neighbour classifier [39]. Variations on this algorithm [36] and performance of other classifiers are discussed in the literature [89,99]. Results suggest the that the SOM clustering approach provides the best overall performance [76]. In this case study, the SOM is used in clustering neural activity into a small number of classes.

A SOM simulator has been coded in *C*, to run on SUN Sparc workstations. The simulator is tuned to the application and tested with various neighbourhood decrease routines, graded gain functions (Figure 4.5) and different distance measures to improve the speed and accuracy of the algorithm. Euclidean distance measure proved to be the best distance criterion. Multidimensional (hypercube) output grid geometries have also been used, but these architectures did not improve the performance, as the number of output classes in this problem is too small to exploit the extra dimensionality and complex neighbourhood relationship introduced.

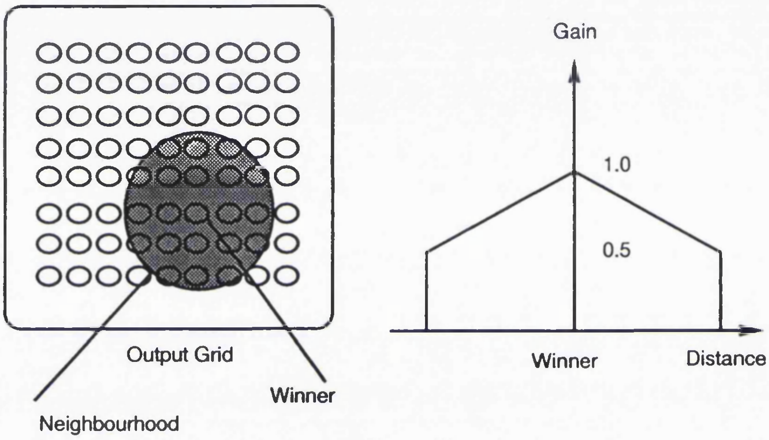


Figure 4.5. Gain and Neighbourhood

The data for the clustering program were recorded in the laboratories of University College London. The datasets consisted of recordings of the 'place cells' and the 'theta cells' from the rat's hippocampus [119]. The spike recordings are 200 voltage values read by a number of electrodes. These 200 dimensional input vectors, are clamped into 200 input nodes. As outputs, the spikes are clustered into a given number of classes. However, the definite number of cells recorded is unknown, up to 10 or 12 cells can be recorded in each experiment. It is necessary to establish the number of output nodes in the network, prior to training the network. As a strategy, it is better to overclassify the spikes, and then after an examination merge these sub-classes, rather than under-classify and face the difficulty of classifying them again. So, usually, a slightly bigger than expected number of output nodes is set as the number of output nodes.

Once the training is completes, the weights are saved, so another set of spikes can be clustered into the currently established classes. This approach can also be used to bootstrap the network, that is, to start training with a set of weights obtained by other methods (such as manual clustering). The training can be supervised, usually to save time, by selecting a typical subset of the input set rather than using all the samples recorded in an experiment. These selected samples are so called 'ideal samples' which would form the basis for the cluster centres.

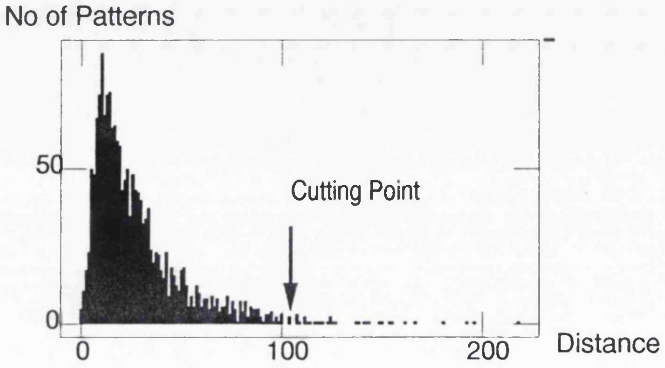


Figure 4.6. Error Histogram in Clustering

During the data acquisition stage, noise occurs in two forms; Systematic Noise and White Noise. Systematic noise often originates from electrical devices close to the laboratories, such as power supplies etc. The network clusters these vectors as a separate class, because of the distinct waveform they possess. White noise is random, and may be generated by various sources such as the disconnection of one of the electrodes, etc. The SOM clusters these noisy input vectors to the closest class available. White noise is undesirable especially during the training stage, as noisy vectors influence the weight update procedure by corrupting the cluster centres with noisy data. To eliminate this problem, a histogram based routine was added to the simulator. This routine calculates the distance of all vectors to the respective cluster centres, and an error histogram is set up for all vectors with the number of bins typically a tenth of the total number of vectors. On this histogram any spike which is further from a certain distance is labelled as noise. This distance is determined by examining the histogram bins and identifying the first bin with zero spikes, after the mean, as the "cutting point" (Figure 4.6). This method is a short-cut measure; the alternative is to calculate the 3-4 standard deviations distance from the mean as a threshold point. All spikes beyond the cutting point distance are labelled as noise and these spikes are prevented from taking part in the training procedure. This process is repeated over each cycle, and result in all outliers being accumulated in a separate noise class.

A feature of the SOM is that it imitates the input vector distribution in the output vector distribution. This poses a problem in the representation of the members of a vector set with a low density (a small number of samples). Uneven distributions pull most of the output nodes into the densely sampled part of the signal space. This problem has been tackled before [38], and a similar procedure is added to the simulator. A limit has been introduced to the weight update procedure; when reached, it stops the weight update for that node, preventing the over-representation of a class of patterns, and allowing less densely distributed patterns to form as a new class.

Another problem related to the input vector distribution occurred in some sets of recordings. In these datasets, the spike distribution showed a tendency to form an elipsoidal shape of distribution, densely packed around a slope, rather than a spherical type of distribution. The Euclidean distance measure assumes a spherically partitioned space and it misclassifies the spike vectors. As a solution to this problem, 6 new nodes are added to the current 200 input nodes. These nodes are clamped with the 6 slope values which relate to each spike's delta values, delta being the difference between the maximum voltage and the minimum voltage read in each electrode. The slope is the ratio of these deltas recorded on different electrodes. By adding this information to the input vectors, 6 new variables are evaluated in the similarity test. The Euclidean distance test that compares the waveforms, also compares these slope values, and spikes are grouped together with respect to their distance in slope angles. It is possible to scale the importance of the slopes by simply exaggerating the Euclidean error derived from these variables. This is done by adding a coefficient to the error calculation that gives a certain weight to the error originating from the slopes.

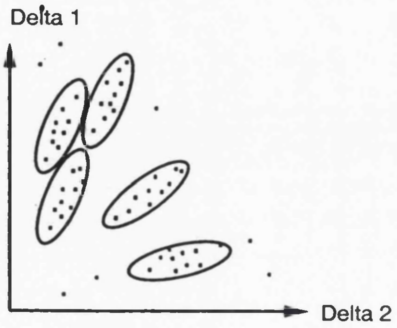


Figure 4.7. Neural Spike Clusters

The results of the automatic clustering with the SOM were compared with the results of manual clustering. In manual clustering, an expert can display the spikes on the computer monitor, and by selecting the best combination of the four electrodes, can visually isolate a group of spikes from the rest. Clusters can then be purified by using histograms. The spikes can be viewed in different combinations of the electrodes (1 versus 2, 1 versus 3, etc,.). But this is a laborious process and it is partly subjective. A numerical comparison between the two methods showed that on a dataset with 2202 spikes, 142 discrepancies were found between the manual and the automatic methods. Of the 142 spikes, 31 spikes were clustered as noise by the SOM, and 47 spikes were clustered as noise by manual clustering, only 64 spikes were placed into different clusters by the two methods. This is a small difference considering the gains made by automating the process.

One measure of success in clustering is the circumference of the clusters. It is assumed that the more compact clusters are, the better the classification is. A statistical analysis on the two clustering methods shows that clustering with the SOM produced more compact clusters. The analysis was done by examining each cluster and measuring the variance from the mean (centre of the clusters) for each of 200 dimensions. Then the standard deviations were calculated for all these dimensions. The mean of these standard deviations for each cluster for both methods are shown in Figure 4.8. Most of the clusters (except cluster 1) obtained by the SOM are more compact, with smaller radiuses to hold most of the spikes.

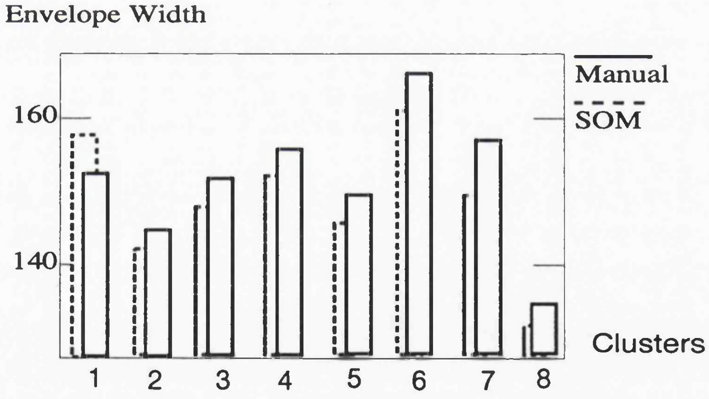


Figure 4.8. Cluster Envelope Widths

Automatic clustering using the SOM gives close results to the results obtained by human experts. The system SOM allows us to specify the thickness of the elipsoid clusters. Problems posed by uneven input vector distributions can be solved. By adding a histogram based noise filtering technique most white noise and systematic noise can be eliminated. These results enable us to do a series of automatic clustering runs and decrease the time spent in the data processing.

4.3.3. Computational Analysis

Currently, the SOM automatic clustering simulator runs on a SUN Sparc station. On this machine, an average training session for the simulator takes about 3 to 4 minutes for 2000 spikes. This is still far from real-time execution where clustering can be done automatically within a few seconds or milliseconds. Once the cluster centres are established, a series of recalls can be done on other datasets to detect any similar spikes. The recall operations are much faster, and on SUN Sparc station they last less than 30 seconds for the same data size. The following operations which involve the calculation and the selection of a winner, are the basis of a recall;

$$Score_j = \sqrt{(I_i - W_{ij})*(I_i - W_{ij})}$$

$$j = \min(score_j)$$

Once the winner is established the weight update is carried out for all the neurons within the current neighbourhood, as follows:

$$\Delta W_{ij} = \eta.(W_{ij} - I_i)$$

As can be seen above, computationally, the most expensive part of the SOM simulation, is the calculation of the scores. In this case, a series of subtractions is followed by multiplications, and finally a square root is taken to establish the Euclidean distance of the given input pattern to the nodes in the output grid. The winner node is established by finding the node with the minimum distance to the given pattern. An alternative to this is to calculate the dot product of the normalised input vector and the weight vector and to select the output node with the maximum scalar product.

The normalisation, the scaling, and the calculation of the score, the sorting to establish the minimum score, take up a sizeable amount of CPU time. In addition to these, in this application, a conventional statistical technique, a histogram routine is also used which consist of multiplication and sorting. Parallel techniques, such as a parallel dot product operation and a parallel sort can be used to speed up the simulations.

4.3.4. Parallel Hardware Mapping

Two cases are considered for the structural parallel mapping of the SOM;

1 - Fine-grained mapping - Each neuron in the output grid is mapped onto a single processor. This enables the system to calculate the score in parallel for all output neurons. Then, all neurons report the result to a central processor which finds the winner. To establish the overall winner, local search and sort techniques can be used and implemented in parallel. In the weight update stage, the supervisor processor sends weight update signals to the neurons within the neighbourhood of the winner node.

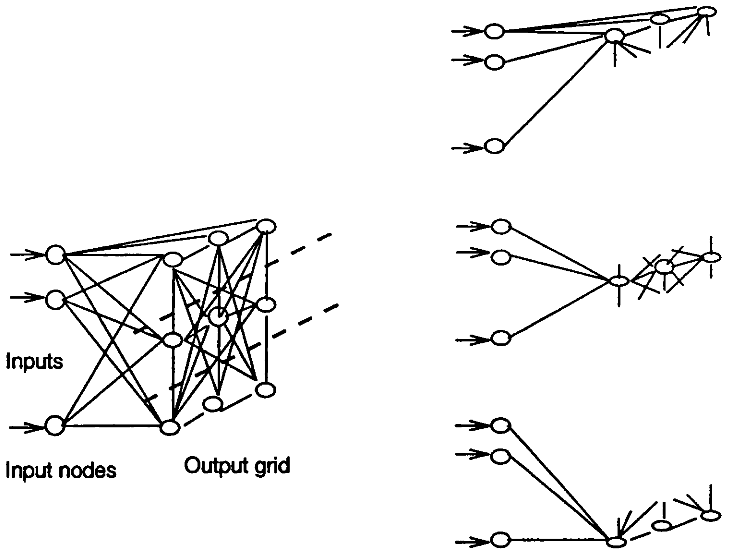


Figure 4.9. Structural Mapping of the SOM Model

2 - Coarse-grained mapping - This approach involves dividing the output grid into sections with an equal number of neurons, and distributing the sections onto a number of processors (Figure 4.9). Some inter-node communication traffic is localised and reduced. However two difficulties remain. The first one is finding the overall winner. This can be achieved by finding local winners, reporting them to the central processor which identifies the global winner and reports it back to the processors. The second difficulty is in expressing the neighbourhood relationship between the output neurons. In weight updates, the standard algorithm has a conditional statement that checks each node to establish whether it is within the neighbourhood of the winner node. On a parallel platform, this would involve the transmission of the neighbourhood definition every time a winner is established. Kohonen later suggested that the neighbourhood can be interpreted as a form of connection weight [92], and this approach is more suitable for eliminating the transmission between parallel distributed processors. The neighbourhood relationship can be treated as a lateral connection weight between output neurons. All lateral weights have symmetrical values between output nodes, which is based on the physical distances of these nodes from each other. Lateral weights could be set at the beginning of the simulation and updated after every epoch to result in a shrinking neighbourhood. Each epoch typically involves a complete dataset presentation and the lateral weight update involves reduction of lateral weights as the training progresses. The lateral weight technique can be useful for mapping the SOM onto parallel hardware in simulations, and for the hardware implementations of the model.

The SOM has a flavour of neural processing, with biological correlates, but it can also be represented using a number of vector, matrix operations. By optimising these operations on fast vector/matrix operator hardware architectures, the SOM can be simulated much faster. In chapter 5, neural network representation issues are considered, and the SOM is programmed using matrix-based representations.

4.4. The Backpropagation Model

The Backpropagation learning algorithm is historically important. It prompted the resurgence in the neural networks field, in the 1980s, following the early disappointments of the late 1960s, prompted by the Minsky and Papert report [114]. They suggested that Perceptrons cannot learn certain associations due to the lack of an efficient learning algorithm to train the weights in the hidden layers. The Backpropagation algorithm was put forward and popularised by Rumelhart [130] and others, answering specifically the question of training the weights to the hidden layers in a systematic manner. The model is based on the Delta rule (also called Widrow-Hoff learning rule), in which the error is back-propagated from the output layer towards the input layer (Figure 4.10).

4.4.1. The Model

The Backpropagation model demands: the choice of a suitable network configuration with a sufficient number of hidden layers and neurons in the hidden layers; the choice of an activation (squashing) function for neurons; the choice of an appropriate learning rate, and a tolerance level, allowing a certain amount of error in the accuracy of the network; and finally, the initialisation of network weights with small random values.

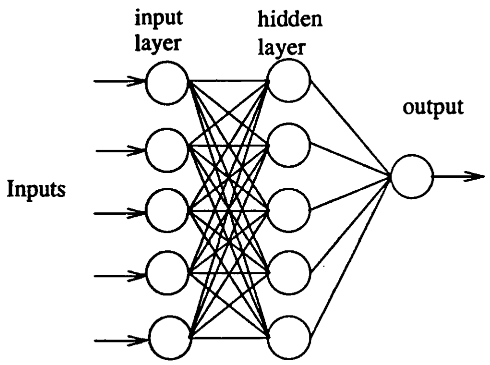


Figure 4.10. A three layered Backpropagation

The training session has the following steps:

1 - A pattern vector is clamped to the input nodes.

2 - Forward Pass - This process involves each neuron multiplying all the input values with the incoming weight values, summing them and passing the resulting value through a nonlinear activation function to produce outputs. This process is carried out for all layers, and finalised at the output layer. There, the outputs of the neurons are the network's response to the applied input pattern.

3 - Error Calculation - The outputs of the network are compared with the targets. Resulting error values are passed through the derivative of the squashing function.

4 - Error Feedback - The error values for the output neurons are propagated backwards by multiplying with the related connection weights.

5 - Weight Update - The calculated and stored error values for each neuron are used to reduce the error of the system.

6 - The processes of forward pass, error calculation and feedback, and weight update are repeated until the error is below the allowed tolerance level.

A number of parameters have a strong effect in the training process, and the resulting internal representation heavily depends on them. These are:

- Learning Rate - The learning rate is a coefficient used in updating the weights. Usually a small value between 0.1 to 0.3 is used, but there are various strategies for optimising the learning rate throughout the execution. If the learning rate is set too high the network might saturate and get trapped in a local minimum. Keeping the learning rate too low results in a long training time.

- Tolerance - the tolerance value determines the amount of error allowed between the outputs of the network and the targets. If the tolerance is too small, the network may never pass this value, and if it passes the network is often overtrained, and it cannot generalise.

- Hidden Units - The number of hidden units in a multi-layer network is important. Having too many hidden units results in an accurate recall with no generalisation. On the other extreme, a network with too few hidden units cannot learn the task.

4.4.2. Case Study: Financial Forecasting

The Backpropagation model is a hetero-associative vector mapping algorithm. The Backpropagation network can learn trends, and provides a good degree of generalisation, which is desirable in financial data processing applications. In this field, the Backpropagation model has been successfully applied to problems such as stock market prediction, risk assessment on mortgages and bond rating [43, 86]. Stock market and currency exchange rate prediction are historically difficult problems for conventional models. Therefore, financial forecasting was used as case study for the Backpropagation model.

A simulator based on Backpropagation algorithm has been developed using *C*, and supported by pattern processing routines to allow time-series handling. The network consists of three layers. The input layer is a string of nodes where the elements of the time series are clamped. The number of input neurons represents the number of past steps scanned in the time series. As a rule of thumb, the same number of neurons are selected for the hidden layer. Finally, the output layer contains a single unit. In training, the next step in the time series; the target is clamped to this node. In recall, this value is generated by the network providing the network's prediction, based on the training. The model is supported by pre-processing and post-processing modules to transform data into formats which produce observable results.

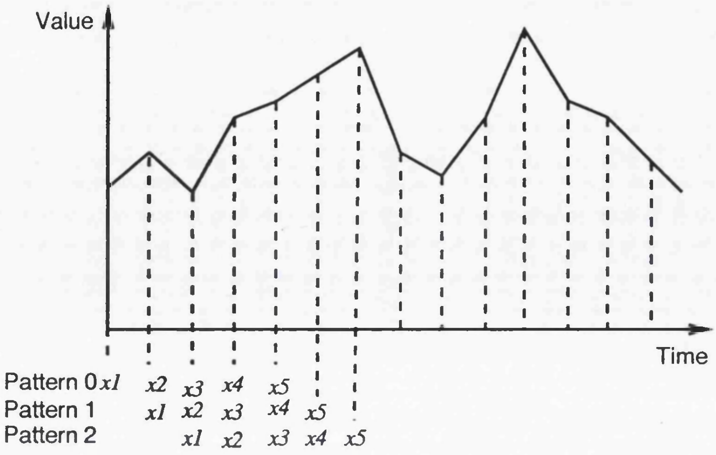


Figure 4.11. Lagged Patterns for the Backpropagation

Pattern Handling - The pre-processing of input patterns takes up an important part of the simulation. The inputs for the simulator are time-lagged values taken from the related time series (Figure 4.11). First, a suitable width for the time window is selected. As an alternative, in some cases, important lags in the time series are identified, and these may be scattered in the time series. When choosing the time lags, an auto-correlation routine can be used to identify the most significant lags in the time series This can be done by correlating elements of the time series with all the other elements of varying time distances. (Figure 4.12).

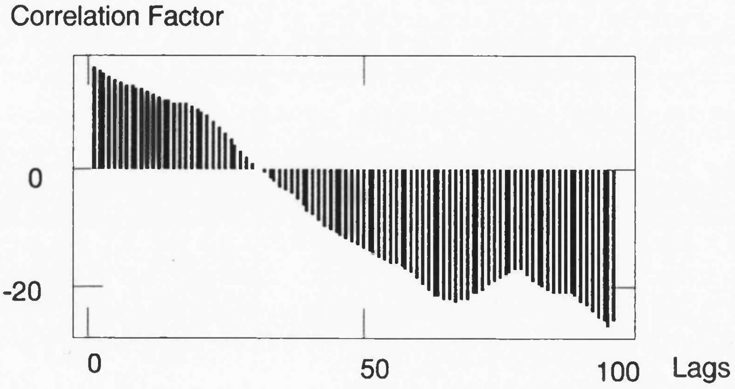


Figure 4.12. Autocorrelating Input Values

The network is provided with normalised or scaled input patterns as the nonlinear activation functions have certain bounds. The patterns can be organised in such a way that the elements of the time series represent differences between the current value and the previous value (this is called differencing). In the post-processing stage, the output vectors are 'undifferenced' or 'unscaled', depending on the pre-processing type applied.

Once a satisfactory level of training is achieved, the network can be used to generate a forecast. Then, the network is provided with a novel input vector, and it generalises from its training. Depending on the pre-processing already done, the output values are post-processed and displayed in the required format. There are two modes of forecasting:

a) Short term forecasting (Recall Mode) - An input pattern is clamped to a trained network, and an output is obtained. By shifting the time window, and repeating this process, a series of short-term forecasts can be made. The simulator can be trained to generate values relating to one, two or more steps ahead.

b) Long term forecasting (Forecast Mode) - Exactly the same steps as for the short term forecast are followed, to generate an output value for a given input pattern. This time the recalled output value is used to form the next input pattern. Another prediction is made and the process is repeated to generate further predictions by using the predictions as inputs.

The learning rate and the number of hidden nodes are important as together they balance generalisation versus overtraining. The correct identification of the most significant lags and the width of the time window are also important in the forecast. The network can be overtrained, or saturated, by choosing a high learning rate and a small number of hidden nodes. This often results in highly accurate recalls, but poor generalisations, and thus poor forecasts. On the other hand a loosely trained network, although not very accurate, generalises much better and produces better forecasts.

Having a greater number of input nodes allows the system to scan larger amounts of data at each step. In this case, computational constraints must be considered, as the increased number of neurons causes the network training to take longer. An optimum network topology must be engineered considering the hardware environment and the problem domain. Other parameters, such as initial random weights, affect the training procedure, and the initial point for a long term forecast also affects the results.

The most versatile use of the simulator is that other data can be presented together with the time series information. These additional data are clamped onto new input nodes in the same way as new input dimensions. As long as these elements are scaled in the same way as the time series data, the simulator treats them in the same way. In financial forecasting, these additional dimensions can be parameters relating to the current political climate, risk factors relating to the economy or even the public opinion polls.

A series of test runs was made on the simulator using real data. The objective was to test the feasibility of the technique and to evaluate the computational requirements for such an application. One dataset consisted of 97 days values of the FTSE 100 index (Figure 4.13). The dataset is divided into two sections; the first 50 days comprise the training set, and the rest of the data are used as the test set.

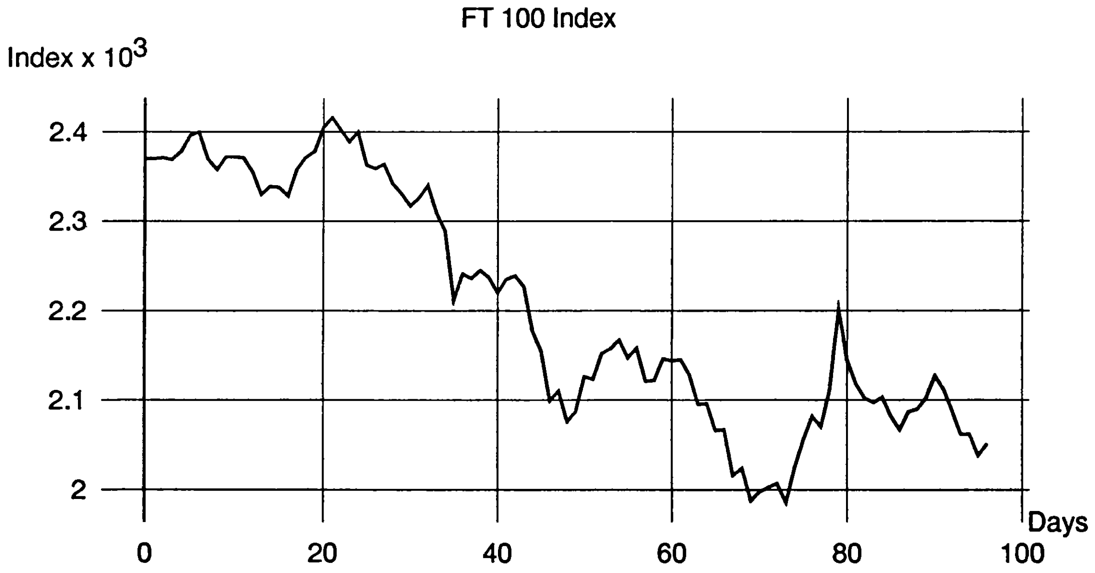


Figure 4.13. The Dataset: FTSE 100 index

An 8 days wide time window is selected; so, the input patterns are 8 dimensional vectors, stretching 7 days into the history of the time series. In this test, two hidden neurons were used, and a single output neuron clamped to the target value, which is the next value (tomorrow) on the time series. The patterns have been differenced and scaled and clamped into the input layer. After about 4000 cycles consisting of all 40 pattern presentations, the error dropped down to an acceptable level (Figure 4.14).

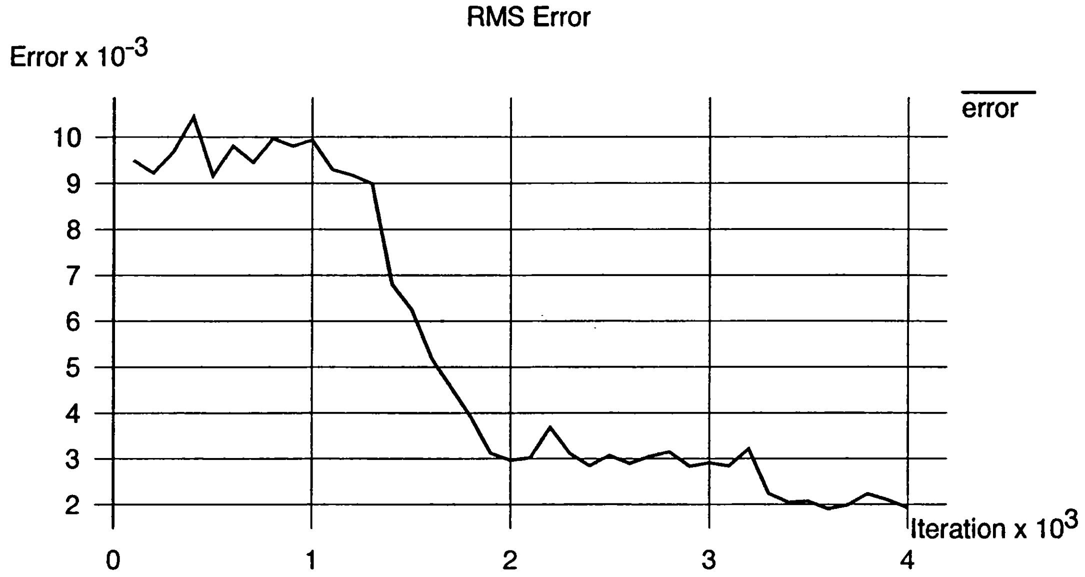


Figure 4.14. Euclidean Error in Training

When tested on the training dataset, the simulator generates very close results to the targeted values (Figure 4.15). It must be noted that, these experiments only show the applicability of the Backpropagation model to forecasting problems. Otherwise this size of dataset is not sufficient for a good generalisation.

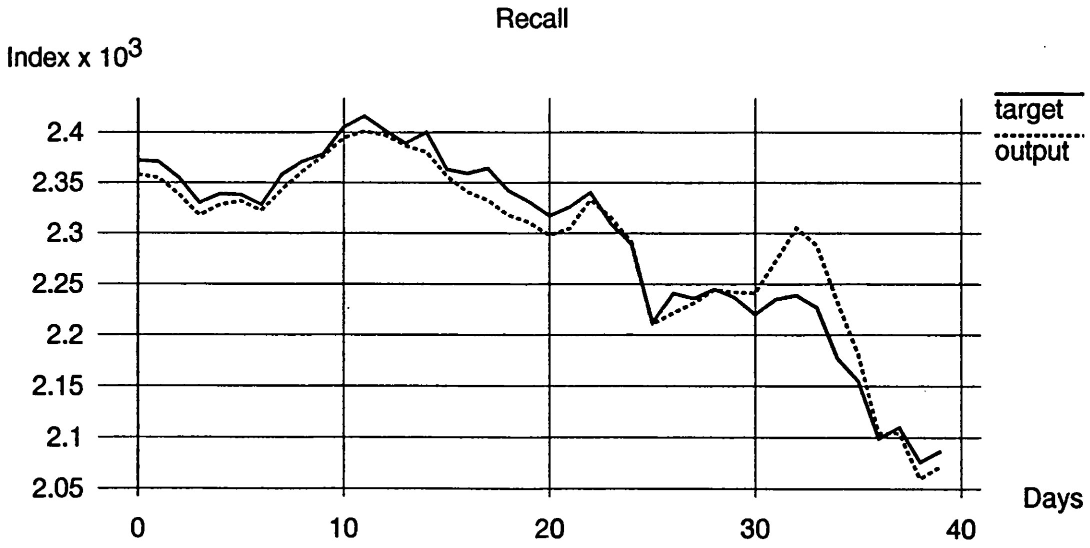


Figure 4.15. Test on Learning FTSE Index

Following this, a long term forecast was made in the second half of the dataset (Figure 4.16). The long term forecast results are particularly interesting as they shows that the network seems to have captured a sinusoidal trend from the first half of the dataset. It is forecasting the same trend for the second half.

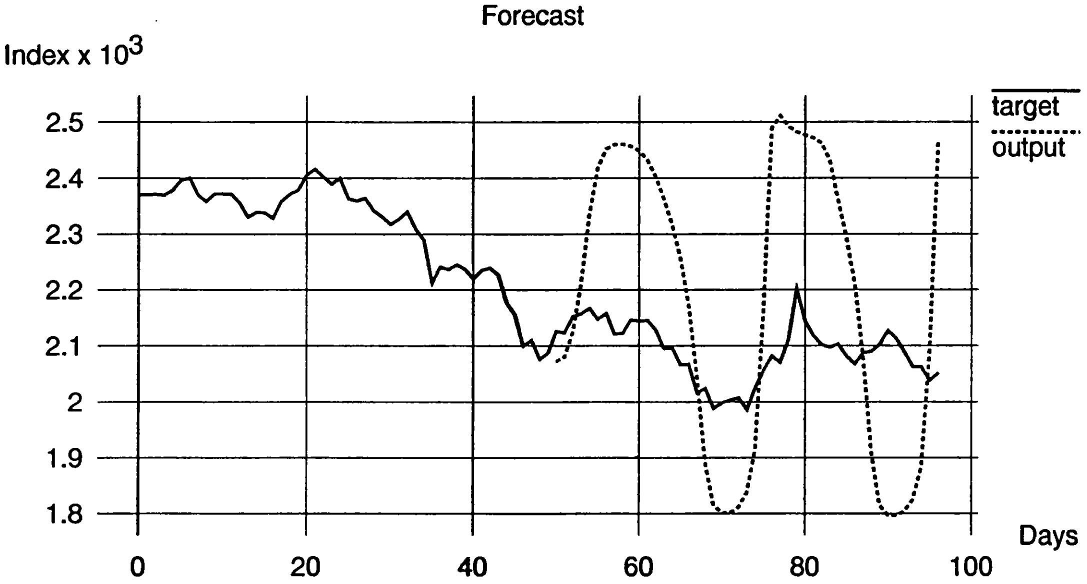


Figure 4.16. Long Term Forecasting FTSE Index

As a result, the experiments show that, the simulator is able to generalise where it has no prior experience. As the training times get longer, the simulator gains higher accuracy. There is a trade off between the number of datasets learned and accuracy in recalling previously learnt datasets. The forecasts depend on the network architecture and the choice of parameters and initial conditions. The simulator behaves very much like a human expert. In the training stage, it builds an internal representation and in forecasting, uses this representation to generate an expert guess. Again like most human experts, the simulator is not able to explain why it has predicted a certain sequence, and it would not be able to predict a stock market crash unless it had experienced one.

4.4.3. Computational Analysis

Real financial neural network applications are computationally demanding. They require the scanning and processing of large amounts of data quickly. Real-time training systems are particularly demanding on computational resources. Because of this, most systems carry out training off-line in batch mode, and execute only real-time recalls.

The Backpropagation training process involves the simulation of the following operations:

Recall:

$$A_h = \sum I.W_h$$

$$S_h = f(A_h)$$

$$A_o = \sum S_h W_o$$

$$O = A_o = f(A_o)$$

Error Calculation:

$$E_o = f'(S_o).(T_o)$$

$$E_h = f'(S_h).\sum E_o W_o$$

Weight Update:

$$\Delta W = \eta.S.E$$

Each operation can be carried out simultaneously for all the neurons in the same layer, but there is a strict sequence of data flow between the layers, which must be followed. Unix 'gproff' results show that, during these simulations, most of the CPU time is spent on multiplications. The nonliner threshold function simulations are also computationally demanding, but they are not repeated as many times as the multiplications. Look-up tables can be set up to approximate the functionality of the activation functions, but special care is necessary in the implementation of these tables. A close look at the sigmoid function shows the rate of change is not uniform throughout this function, with a near-linear transition in the middle, and a strong nonlinear character at either ends of the function. This characteristic must be preserved on the look-up table representing the function.

In financial forecasting the Backpropagation model is computationally intensive on the training stage depending on the data and network size. On the SUN Sparc workstations, most practical financial forecasting networks can be trained within a couple of hours. In extreme cases, overnight executions might be necessary due to increased dataset size, such as the last 10 years daily index of a financial indicator. Although data can be processed by the network without human intervention, it makes sense and saves time to carry out a cluster analysis on data to purify and obtain a smaller representative dataset. A simple data scan can eliminate unusual or corrupted patterns to save time, preventing the network from spending hours on a difficult sample which is impossible for it to learn.

Usually, neural networks in financial forecasting are not large, the shifting window technique is sufficient to process data from the last few days or a month, to make a short term prediction (for the following day). The FTSE prediction system needed a network with 30 input, 30 hidden and 1 output neurons. The pre-processing and post-processing operations and graphics display functions are also computationally significant. These non-neural computations cover: finding a maximum or a minimum for an array, normalising, scaling, transforming, and the graphics displaying of the patterns. The experience in many simulations is that the graphical tools cannot keep up with the neural network part of the simulator.

4.4.4. Parallel Hardware Mapping

Two cases are considered for mapping the Backpropagation model onto parallel hardware.

1 - Fine-grained structural mapping - Assuming there are the same number of processors as the number of neurons in a network, and as many communications channels are available as the number of connections in the network, each neuron can then be mapped onto a single processor. A central processor can supervise the parallel execution, and check whether the tolerance test is passed. The outcome of this approach is massive interprocessor traffic particularly between the processors for the hidden layers and the output layer. An alternative to this is to accumulate small weight changes and carry out a batch of weight updates after all the patterns are presented. In fact, the batch weight update techniques are frequently used to reduce the communications traffic, and achieve efficient mapping on parallel hardware.

2 - Coarse-grained structural mapping - Considering that there are less processors than neurons in the system, two types of structural partitionings and subsequent mappings can be carried out; layer partitioning and network splitting.

Layers of a Backpropagation network represent concentrations of similar computations on data. All operations taking place within a layer can be executed simultaneously, as data for each layer are presented at the same time. For example, a three layered Backpropagation network can be mapped onto a two processor parallel system with a host. The host would undertake the I/O operations and send patterns to the first processor which is the hidden layer. The hidden layer calculates the states of the neurons and passes them onto the output layer. When the output layer is calculating its states, the hidden layer can continue with the next input pattern. The overall process would be a data pipelining process with performance heavily dependent on the interprocessor communications speed and bandwidth. The pipelining approach could be effective where batches of recalls or forward passes are carried out, keeping all processors busy. However, this method cannot provide any speed up in a real-time recall operation where an output pattern is desired following a single pattern presentation.

The second approach involves splitting the network in a horizontal line (Figure 4.17). This type of mapping creates a data parallel execution and parallel processing takes place on all processors. There are two problems with this approach. Firstly the Backpropagation algorithms is a so-called non-local learning rule, where the weight values depend on parameters other than the immediately connected neuron states. This necessitates the transmission of the state or error values to all other neurons. In the case of network splitting these values must be transmitted to other processors to measure success globally. This introduces inter-processor communications. The second problem with this mapping technique is that a 100 % load balance between the parallel processes must be achieved for an efficient parallel executions where no processor stays idle. This may not be possible all the time, considering networks with odd number of neurons.

Another parallel mapping approach which is useful for training but impractical for single pattern real-time recall, is called pattern parallelism or training parallelism. This approach prescribes the mapping of complete nets onto each one of the many parallel processors. All networks train on different subsets of patterns, occasionally updating the common weights through a communications mechanism.

The Backpropagation model can be seen as a multi-dimensional feedback control process with the following functions. Firstly, the input vectors are multiplied by weight matrices, and the resulting vectors are transformed by a nonlinear function. On the consecutive layers the same operations are repeated, and finally an output vector is generated as the response of the system. The output vector is subtracted from a target vector, an error vector is generated and then multiplied by the connection weights vector and the result is transformed by the derivative of the nonlinear function. Error vectors calculated by this process are used to modify the weights matrices.

The biological plausibility of the Backpropagation model is extremely unlikely. It is more plausible to think of it as an adaptive feedback control mechanism, modifying internal system parameters to achieve desired outputs. This kind of thinking is more

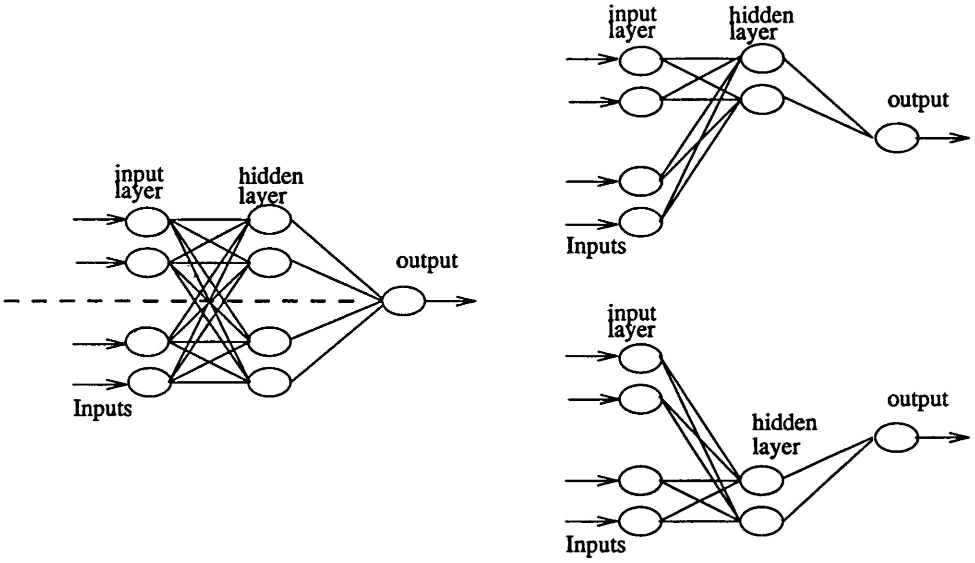


Figure 4.17. Structural Mapping of the Backpropagation

liberating as it puts the emphasis onto the actual computations rather than the neural philosophy. The computations involved in the simulations of the Backpropagation model are a set of vector/matrix operations. By optimising the execution of these operations the Backpropagation model can be executed efficiently. This thesis strongly supports the view that the Backpropagation model is a vector-matrix based algorithm with a large fanin/fanout. In chapter 5, the model is programmed in matrix/vector based representations, and in the following chapters, these representations are mapped onto parallel hardware.

4.5. Comparison of the three Models

The three models differ in their training and recall procedures, in their applications and computational requirements. They show structural differences which are important for parallel hardware mapping. Table 4.1 shows a comparison between these models in terms of their structure, learning and recall procedures. In this table, most calculations are vector arithmetic operations with the exception of activation functions (*f*) and their derivatives (*f'*).

The main strength of the Hopfield model is that the Hopfield neurons can operate synchronously or asynchronously, independently of each other, on binary or continuous values. The network can be used in optimisation tasks, producing a good solution in a short time. The main weakness of the Hopfield net is the setting up of the weight matrix.

Model	Structure	Learning rule	Recall procedure	Error calculation
Hopfield Nets	Each node connected to all others	$W=\sum I'.I$	$S_i=f(I_0+\sum S_j W)$	$E=\sum I-S$
Self-Organising Map	Input nodes fully connected to Output grid	net $- \sum I.W$ $j_{winner}=\max(net_j)$ $\Delta W_{ij} - \eta.(I_{ij}-W_{ij})$	$net=\sum I.W$ $j_{winner}=\max(net_j)$	$E=I-W$
Backpropagation	Multi-layer Perceptron	$\Delta W_{ij}=\eta.S_i.E_j$	$net=\sum I.W$ $S=f(net)$	$E_{jo}=f'(net).(T_j-S_j)$ $E_{jh}=f'(net).\sum E.W$

Table 4.1. Neural Networks Structure and Properties

Even in the case of successful setting, the number of patterns that can be stored is limited. In addition to this, Hopfield nets tend to fall into local minima.

The main strength of the SOM is that it is an unsupervised learning algorithm driven by input data. Real world data can be presented as training data, with the network adapting itself to changes in the inputs. The parameter selection process for the SOM is the main obstacle in the design of the applications.

The strengths of the Backpropagation algorithm are proven by its popularity and its wide use on a range of real world applications. The major weakness of the Backpropagation model is the initial design of its architecture, which is a problem-dependent operation involving a series of tests and experiments. In fact, the design of the network and the selection of the parameters is more of an art than a science.

Research is continuing in optimum neural network topology and parameters, fast learning algorithms and automatic network design. One way of computationally tackling this problem is to use multiple neural network architectures.

4.6. Multiple Neural Networks

A large number of real-world applications require the use of hybrid neural network architectures due to the heterogeneous nature of the problem data. Robotics applications demand the extraction and processing of multiple sensory information, and the use of this in simultaneous control tasks [93]. In financial forecasting, many different financial indicators affect a particular time series. The indicators can be analysed separately and the results can be incorporated in the final prediction system. In other applications it makes sense to divide tasks into smaller ones and to use a number of simple networks [31, 100, 105, 106].

The theoretical limitations of understanding a massively large number of interacting elements is hindering neural network development. Also, it is widely known that large neural networks computationally do not scale-up. In addition to that, some models are suitable for solving certain problems. Modular architectures can enhance the performance of models by integrating these models so they complement each other [69, 75, 139]. The Nestor Learning system [4, 127] was an early example of using multiple neural networks in pattern recognition problems.

Network level competition can be used to achieve automatic task decomposition [80, 81]. Jacobs and others [80] demonstrate the use of modular architecture and automatic task decomposition. They focus on a major problem with the Backpropagation model, the so-called 'temporal crosstalk' problem. This problem is manifested as the inability of the Backpropagation network to learn patterns which produce conflicting information in the hidden units. A fast learning has been demonstrated through automatic task decomposition on a modular architecture. Similar approaches have been used in classification problems [29, 30].

Optimum neural network design is another time-consuming problem. Choosing the correct topology and initial parameters for a network is itself a NP-complete problem. The complexity is increased especially if general solutions are required for a wide range of problems. Again, network competition can be used to achieve optimum networks for a specific problem. A large number of networks can be initialised with different parameters and the ones which reach a convergence with a good generalisation and robust performance are chosen. This approach is close to a new computational paradigm; Genetic Algorithms which itself is a candidate for solving pattern recognition and optimisation problems.

The brain, which is the most advanced computational device known to humans, is not homogeneous. In fact, it portrays a modular architecture with task partitioning which is noticeable at the highest level with its two hemispheres. Different tasks seem to take place in different parts of the brain. PET (Positron Emission Tomography) scans and MRI (Magnetic Resonance Imaging) results confirm this claim. Biological evidence from the brain encourages task decomposition and modularism at network level with inter-module cooperation and competition. There is neither the technology currently available to build machines which have millions of interacting processes, nor is there the expertise to program machines with such complexity. Technological limitations force the design of modular software and hardware.

Modular designs and task decomposition can be used to argue for the design and development of Hybrid Systems. These systems will be programmable and evolvable, and will exploit intelligent knowledge-based systems, neural networks and genetic algorithms combining the best features of all three models [9]. For example, neural networks are particularly good at knowledge extraction which is the main problem for building knowledge-based systems. An interesting combination would then be to use neural networks as the front-end, in knowledge extraction and elicitation for expert systems [96]. Genetic algorithms can be used to design optimum neural network architectures through evolution [58, 133]. General-purpose parallel hardware platforms can be used to achieve a parallel evolution in the automatic generation of neural networks.

An efficient mapping strategy must consider these trends in algorithm research, and support modularism and parallelism in the four levels of neural network execution: the application, model, representation and execution environment domains.

4.7. Conclusion

In this chapter, in the form of three case studies, three neural network models and their overall computational requirements have been analysed. This approach has proved to be more informative compared with exclusive algorithm analyses. The three neural network models examined here, show differences in their applications, training and recall procedures, their structural properties, and in terms of parallel mapping and hardware implementations.

The Hopfield nets are suitable for auto-associative recall problems. Research is necessary on this model to find learning or data storage algorithms to fully exploit its potential. The Backpropagation model has proved itself in many commercial applications. In training, it requires great computational power for realistic applications. The SOM is a good statistical tool to detect salient features in datasets, automatically clustering patterns with no prior knowledge about them.

Structural parallel mapping examples shown in this chapter reveal that there is not a single method applicable for all neural network models. The Hopfield and SOM neurons are homogeneous, and they can be grouped arbitrarily and mapped onto parallel processors with varying granularity. The complexity is increased in the case of the Backpropagation model as different partitioning techniques result in varying load and communications requirements. The difficulty of designing a generic mapper for a wide range of neural network models is stressed in the variation of structure and properties during the evolution of these models.

In terms of hardware mapping and implementation, the Backpropagation model is the most complex of the three models. Its multi-layer structure makes it difficult to implement on silicon. Neuron by neuron implementation on silicon would be relatively more expensive due to the complexity of the Backpropagation neurons. Both Hopfield nets and the SOM have already been implemented onto silicon, and this trend will continue as the relative cost of hardware implementations is reduced.

The most common aspects in all three models, is that all computations involved are vector-matrix based operations. The Hopfield net is an association matrix acting upon noisy, corrupted or incomplete input vectors, settling to a state which represents the original vector. The Backpropagation model provides hetero-associative mapping between input and output vector sets, achieving this by a series of weight matrices between consecutive layers. Finally the Self-Organising Map topologically orders sets of input vectors on an output grid.

Real world applications involving the three models require conventional computing routines such as, normalisation, scaling, histogram and graphics display and input/output procedures. These routines sometimes require greater computational power than the neural network simulations they support. Again these routines are also vector-based operations involving parallel arithmetic, search, and graphics display operations. A General Purpose Neural Computer architecture, optimising the execution of a set of matrix operations would be valuable in neural network simulations. Chapter 5 presents *MATLIB*, a matrix-based library, as a step in this direction.

Chapter 5

Neural Network Representation

In this chapter, neural network representation and programming issues are discussed. Function-oriented, object-oriented and vector-oriented representation techniques are analysed in terms of their ability to capture neural network properties and mapping onto general-purpose parallel hardware. Based on the analysis, matrix-based C libraries MATLIB and NETLIB are designed and developed to meet these requirements.

5.1. Introduction

Most neural networks today, are simulations on sequential conventional computers, and some simulations run on parallel hardware. Neural network simulations involve a two-stage mapping process; firstly the representation of an application as a computer program, and secondly mapping this representation onto hardware. Neural network programming languages serve as a medium for these two tasks. The efficiency of mappings and the performance of executions strongly depend on the choice of the neural network simulation language. A good simulation language facilitates the programming task, provides easy access to data and methods, and can be easily mapped onto parallel hardware leading to an efficient execution.

Application-oriented and algorithm-oriented programming environments often employ a single programming language for the two mapping tasks. Because of the difficulty of obtaining generalised features in one programming language, some environments adopt multiple representations. General programming environments which aim for generality and flexibility, with high performance, often use two-level representations, employing a high-level language and an intermediate-level language.

An HLL serves as a user-friendly programming domain easing the process of neural network application representation. HLLs are usually supported by graphics application builders and graphics monitors. An algorithm library containing parameterised models often helps in the development of applications. Graphics-based and menu-driven systems appeal to users. They provide a bidirectional medium for neural network programming, as all actions are answered by graphics-based reactions, easing the task of mapping a world problem onto a computer. One disadvantage of graphics-based routines is that they are computationally demanding, and they can slow down the execution on most hardware. What is required from an HLL is that it captures neural-network-oriented features and provide user-friendly access to models and applications.

The ILL representations on the other hand, determine the efficiency of the potential execution. They are used in debugging, monitoring and control tasks during the execution, and also provide an interface between programming and execution environments. The two-level representation strategy introduces a complication, in that the ILL representation often has to be automatically generated from the HLL representation by a compiler, optimiser or translator. ILL representations are either directly compiled to executable code, or a cross-compiler can be used for execution on different target architectures. Hardware mappers exploit the ILL representation, so the efficiency of the mapping and execution directly depends on this low-level representation. Ideally, an ILL, or a low-level neural network representation supports a range of hardwares without losing efficiency. The following considerations are important in the choice of a low-level neural network specification language:

- *Machine independence* - The specification language should be easily ported and executed on a number of different target machines.

- *Flexibility* - The representation should be easily modified and it should meet the requests from the HLL.

- *Clarity and Modularity* - Neural network features coded in the representation must be easily accessed by the user for debugging, monitoring and control purposes.

- *Parallelism* - The representation must support implicit and explicit control for parallel execution.

- *Efficient execution* - The representation must facilitate a fast execution on different hardwares.

High-level or low-level, there are two alternatives in the design of a neural network programming language. Either a new programming language is developed and promoted, or a popular programming language is extended with neural-oriented features. The design and promotion of a new language is not desirable as it suffers from the "yet another language syndrome". Users are reluctant to learn a new language for a new form of computation. This is the reason that most general programming environments use subsets or supersets of current languages such as *Pascal*, *C* or *C++*. Using *C* is the most popular approach, widely adopted for its flexibility, availability, and its low-level features which makes it suitable for exploiting hardware efficiently.

In this chapter, three representation techniques used in neural network programming, are analysed with simulation examples. The analysis focuses on the ability to capture neural network properties, generality, flexibility and the ease of mapping of these representations onto general purpose parallel hardware. As result, a matrix-based *C* library *MATLIB* is proposed, designed and implemented, and the three models are programmed using the library functions. Using *MATLIB* functions as building blocks, *NETLIB* library is developed which consists of the learning and recall functions of the three algorithms. As part of *MATLIB*, a number of data communication routines are provided to enable parallel simulations and test mapping strategies in the course of this thesis work.

5.2. Neural Network Representation Techniques

Three different representation techniques can be used in neural network programming. These are:

- function-oriented,
- object-oriented, and
- vector-oriented representations.

Function-oriented representations primarily focus on the functionality of algorithms. They are the extensions of the classic algorithmic way of thinking to computer programming. Function-oriented philosophy has been popularised by a number of procedural languages such as *C* and *Pascal*. Task break-down and parallelism are feasible within these representations, considering that programs are a list of functions or procedures scheduled by a flow chart. Function-oriented representations are useful in the algorithm development stage, but they deny access to the low-level and fine-grained features of the algorithms which are necessary when mapping onto parallel hardware.

Object-Oriented Programming (OOP) philosophy is one of the recent and most powerful trends in general computing. It is based on the idea that the world is made of self-contained objects, with their own methods, and computer languages should preserve this structure. Currently, *C++* is the most popular object-oriented programming language. It provides a set of classes, as a basis for objects comprising data and methods. If a design follows the object-oriented philosophy, the resulting *C++* programs are modular, easy to modify and upgradable. The programming task for neural network models is easier using an OOP language as a high-level language, and the resulting representation also reflects the neural network structure. *C* data structures can also be used in an object-oriented manner. An example of this is presented in the section with simulations using Pygmalion's *nC system* data structure. In terms of parallelism, OO representations have one major drawback; current hardware architectures are not object-oriented parallel architectures; they cannot match the fine level of neural network granularity with a large number of communications channels between parallel processors. The lack of communications facilities hampers the mapping efforts onto hardware.

Vector-Oriented representations emerged in computing to exploit vector processors and general purpose parallel hardware. Recently, some general purpose, high performance commercial programming systems have used these representations in their low-level languages. An example is the **SKY** system provided by Sun Microsystems [114]. This system provides a set of low-level vector/matrix-based instructions in a library, which are executed on fast vector-matrix multiplier parallel hardware. Applications are written in a high-level language such as *C* or *FORTRAN*, and they are processed by a compiler/optimiser which identifies loops, and generates the vector-based intermediate level representation. Executions of these vector arithmetic operations are optimised on the target architecture. The **SKY** system is also an example of a two-level representation which is used in general computing.

In the following sections, programming examples of function-oriented, object-oriented and vector-oriented neural network representations are demonstrated. The advantages and disadvantages of each approach are highlighted in terms of their flexibility, ease of mapping onto parallel hardware and feasibility of automatic code generation.

5.2.1. Function-Oriented Representations

This philosophy views algorithms as a list of functions or procedures which can be further subdivided into simpler tasks, until they are directly represented by the instructions of the simulation language. This approach is not concerned with the way data are represented, as its main focus is on the methods of processing data.

Typically, neural network data, such as states of artificial neurons, connection weights between the neurons, input and output patterns, are represented as one or two dimensional arrays. These arrays can be global or local variables, depending on the language and programming technique used. Local variables and modular designs result in programs which can be easily modified, expanded and debugged. Most application-oriented and algorithm-oriented neural network programming environments use *C* or *Pascal* based languages with added neural-oriented features. *C* algorithm libraries with a number of customised applications and a set of parameterised functions, provide a sound basis for neural network programming. *C* is widely available, well known, provides low-level features, and is close to the Unix operating system which most high performance workstations run. *C* is used in all the simulations carried out in the structural analysis of the neural networks in chapter 4. As an example of programming a neural network model in *C*, the recall phase for the Hopfield model is presented below:

Function-Oriented Programming with *C*

Although there are many ways of achieving the same result, to program a neural network in *C*, typically a pseudocode of the algorithm is written. The pseudocode for the recall stage for a Hopfield net is as follows:

1 - Allocate memory for input and output patterns, and weights.
2 - Read the fixed weight matrix.
3 - Read an input pattern.
4 - Update the states of all neurons until they are stable.
5 - Output the states.

The fourth step is the core of the Hopfield algorithm. It involves the following operations; an input vector is clamped to neurons, and all states are updated by multiplying the current outputs of all other neurons with the connection weights, summing the results and applying a threshold function to the sum. Then convergence test comprises the comparison of the current states with the previous states of all neurons. When convergence is reached the states are reported as outputs.

A pattern recall function written in *C* is shown in Appendix A.2. In the listing in Appendix A.2., the *dot_product()* function can be executed in parallel for all NEURONS. in addition to this, the *for* loop in the function can also be parallelised. Finally, the *threshold()* function can be executed simultaneously for all NEURONS. Task parallelism is the natural form of parallelism in function-oriented representations. Large problems can be divided into smaller tasks, and tasks which can be executed in parallel are identified. By mapping these tasks onto separate processors, a parallel execution can be achieved. If *C* procedures are highly interconnected units with little communication with other procedures, a small interprocessor communication requirement can be expected. But, as the writing of the *C* procedures are subjective and user-dependent, there is no guarantee that any task partitioning operation will result in low interprocess communications.

5.2.2. Object-Oriented Representations

It is widely held that neural networks are hierarchical structures, and this structure can be reflected in computer programs by using similar data structures. Data structure based representations are object-oriented representations in terms of their emphasis on data, rather than functions. One major difference is that object-oriented representations do not necessarily have a hierarchy, unless specifically enforced. The initial design of the data structure is important, and a range of neural network models must be examined to build a generic structure. This structure must cover all models in terms of their connection topology, data and functionality. The main advantage of this approach is that, independent of the neural network model, the data structure remains the same. This makes the monitoring, debugging and control tasks much easier. An independent graphics program can monitor the data structure during execution and display its data. Functions at any level of the hierarchy can be fired by using a graphics based execution monitor. The low-level access to data and functions is advantageous if neurons or synapses need to be mapped onto fine-grained parallel architectures. Neuron structures can be accessed and isolated as independent objects with their particular data and methods.

The Pygmalion programming environment developed a machine-independent intermediate-level language, *nC*, for the low-level neural network representation [18, 147]. *nC* is based on a hierarchical data structure called *system* which is presented in chapter 3. Object parallelism is possible with *nC* and, as part of the mapping investigations, an automatic low-level object generator **code_gen** has been developed. This program scanned the already initialised *nC* structure and generated all initialised data structures in an ASCII program listing called *nC_code*. This list of *C* data structures can be compiled and executed using a *C* compiler, or the data structures can be potentially mapped onto a fine-grained parallel hardware. The *nC* data structure representation is a low-level representation and was designed to be generated by translating from a high-level language.

Object-Oriented Programming with *nC*

Programming an application with the help of the Pygmalion Graphics Monitor is a simple task. It involves the calling of the parameterised algorithms from the ILL library and initialising the data structures. Programmming or modifying an algorithm, and manipulating the *nC* data structure, on the other hand, is not a trivial task. The main difficulty lies in the setting up of the pointers in the hierarchical tree and allocating memory for corresponding neural network connections. Programming with the *nC* data structure is a low-level programming task. It first involves the identification of the lowest level objects with common functionality. These tasks could be a state-update task for a layer, or a weight-update operation for all the synapses in the network. Then, a *C* function is written following the *nC* parameter passing convention. Finally, the function is placed on the hierarchical tree, by setting up the relevant function pointers to it. When the system is prompted for execution in the highest level, the pointers activate each other hierarchically, and the *C* code is executed.

Appendix A.3. shows a function which is written to decrement the distance and the gain in the SOM. This rule is a simple function that has been made complex by the use of the *nC* data structure. It can be argued that *nC* was not meant to be for code writing, and it was designed to be automatically generated from the high-level language *N*. But it was unavoidable to code in *nC* in the algorithm development stage for the Pygmalion project. In practice, the translation requirement from *N* language, also meant that *N* had to adopt the same hierarchical structure, thus making it similar to *nC*. In terms of parallel mapping, the *nC* representation targeted fine-grained, neuron-based parallel hardware architectures, and it was not suitable for matrix-based general-purpose parallel hardware. It would require additional modules such as optimiser/vectoriser to extract vectors and matrices from the data-structure representation. Only explicit parallelism was accommodated within *nC* which was applied only to repetitive operations such as for and while loops.

5.2.3. Vector-Oriented Representations

The development of vector-based neural network programming languages is a natural step, given the fact that most neural network simulations involve arrays of patterns, states and weights, and vector operator hardwares are fast, high performance devices. Vectorisation would ease the conceptual mapping of algorithms to code, and vector/matrix-based representations would be efficiently executed on parallel hardware. Defenders of the matrix representation philosophy, hold the view that neural networks are matrix-based computational paradigms because of the large fanin/fanout and the fine granularity they possess. This is particularly true in the case of the most popular neural network algorithm; the Backpropagation model, which was analysed in chapter 4. Other neural network algorithms also involve vector arithmetic operations which can be represented in a vector/matrix-based language.

As mentioned in chapter 3, the Galatea Neurocomputing project has adopted this philosophy, and developed a matrix-based intermediate-level language, *VML*, with the main objective of exploiting matrix-based general-purpose hardware boards efficiently.

Vector-Oriented Programming with *VML*

The same steps are followed in the programming of neural network models in *C*. A pseudocode is written, which is based on the functionality of the algorithm, and *VML* rules are written using *VML* I/O, control and execution statements. The resulting code is a compact, *C*-like program which is easy to code. The coding of the Hopfield, the SOM and the Backpropagation models took a short time. These *VML* based simulations ran successfully, demonstrating the viability of the technique. An example of *VML* listing is shown in Appendix A.4.

The main weakness of *VML* is that it has to accommodate all user requirements in the language. Therefore, it is in continuous competition with a language like *C*. The parser and the interpreter have to be modified every time a new statement is introduced. As the *VML* is intended to be generated automatically from *N*, access to data is not straightforward from the high-level object-oriented language in which applications are originally coded. To extract the real-world data, a strategy and a symbol correlation table must be used every time data is monitored. In all aspects, the *VML* representation is a procedural representation with data grouped together in a tightly coupled vector form for the convenience of fast execution. The neural concept about the algorithm is totally lost in this representation. As there are no concepts such as a neuron or synapse, this representation is not suitable for mapping neural networks onto special-purpose, neuron-oriented, massively parallel architectures.

5.3. *MATLIB* Matrix Library

The analysis of neural networks in chapter 4 indicated that the most common operations between the three models can be captured in vector-matrix operations. These operations were mainly vector arithmetic operations with some neural network specific functions applied to vectors. One of the goals of this thesis has been to achieve a generic representation which is: (i) capable of capturing neural network properties, and (ii) suitable for general-purpose parallel hardware mapping. Here a matrix-based *C* library, *MATLIB* is proposed to meet these requirements.

An efficient way of developing neural network simulations is to use parameterised libraries. Source libraries are particularly valuable as they are open, modular, flexible and expandable. Using *C* as the source language makes libraries accessible to other programmers, preventing the repeat of the similar programming efforts. Users can build their applications easily by calling these routines in their applications. As mentioned in chapter 2, *C* Algorithm Libraries provided as part of the major neural network programming environments are the most popular programming tools. Another strength of using the library functions is that it is fundamentally an object-oriented approach. Library functions are multi-purpose modular units communicating with the external world through a list of parameters.

Simulations in *VML* (Version 1.0) show that the language is simple, clear and compact representation of neural network models. The experience with *VML* showed that programming neural networks, using matrix arithmetic operators, is easy and conceptually acceptable. The language can be used for other programming domains with fine granularity. Furthermore, the *VML* functions can be optimised for general-purpose parallel architectures, and executed with a high performance. The main problem with *VML* is that it is a new language with a syntax, parser and a set of operators. A set of matrix-based *C* functions would have been a better choice for neural network and general-purpose programming.

MATLIB library functions have been designed exactly for this reason. *C* is chosen as the library language for its wide availability and accessibility. *MATLIB* functions are *C* routines, and can be modified and upgraded by anyone with the knowledge of *C*. The functions are vector, matrix operators, neural network specific functions and data communication statements. A complete list of the *MATLIB* functions and their arguments are presented in Appendix B.

5.3.1. *MATLIB* Functions

An incremental approach has been used in the development of *MATLIB*. First, the common routines have been developed which are sufficient to represent the three models; the Hopfield, SOM and Backpropagation. *MATLIB* programs for the three algorithms are compact, clear representations which are easy to modify and update. The listings of these algorithms are provided in Appendix C. The *MATLIB* functions are divided into four categories;

- data operators
- arithmetic operators
- neural operators
- communications operators

The data operators are memory management, input/output, data assignment and copy related functions. Examples are, matrix definition (*matdef*), matrix copy (*mcpy*), vector (or row) copy (*vcpy*), column copy (*ccpy*) operators. These operators are handled in a different way on different hardwares. For example, on conventional systems a matrix assignment operation can be carried out by passing the pointer to that matrix; the same operation would involve transmitting the matrix data on a distributed memory multi-processor system. The use of data operators in the three models is shown in Table 5.1.

Mneumonic	Description	Hop	SOM	BP
matdef	matrix definition	✗	✗	✗
matld	matrix load	✗	✗	✗
matsv	matrix save	✗	✗	✗
matsh	matrix show	✗	✗	✗
vcpy	matrix rows copy	✗		✗
mset	matrix element set		✗	✗
sval	get matrix element		✗	
mtra	matrix transpose			✗
mran	matrix randomise		✗	✗

Table 5.1. The use of *MATLIB* data operators

The arithmetic operators used by the three algorithms are; matrix addition (*madd*), matrix subtraction (*msub*), matrix multiplication (*mmul*) and a number of submatrix versions of these operators, which can operate on submatrices within given row or column references. The use of arithmetic operators are presented in Table 5.2.

Mneumonic	Description	Hop	SOM	BP
mmul	matrix multiplication	✗		✗
vadd	matrix row addition		✗	✗
msub	matrix subtraction	✗		✗
vsub	matrix row subtraction		✗	✗
memu	matrix element multiplication			✗
mmax	matrix maximum			✗
mmin	matrix minimum		✗	
mabs	matrix absolute			✗
mavg	matrix average		✗	✗
mscm	matrix scalar multiplication		✗	✗

Table 5.2. The use of *MATLIB* arithmetic operators

In addition to the above, a number of functions are developed to match the so-called 'neural' functions of the three algorithms. These are: application of tangent hyperbolic or sigmoid type of logistic functions, or their derivatives to elements of a matrix, calculation of RMS of a matrix (*mrms*), and the SOM specific lateral matrix update *mlat* function. Table 5.3 shows that the number of neural operators required to program the three algorithms is relatively low, and this is an advantage in the realisation of these functions on the potential hardwares.

Mneumonic	Description	Hop	SOM	BP
mtan	apply tanh to matrix	✗		✗
dtan	apply derivative tanh to matrix			✗
mrms	matrix root mean square	✗	✗	
mlat	lateral weight update for SOM		✗	

Table 5.3. The use of *MATLIB* neural operators

The aim of this separation of data, arithmetic and neural operators, is that these different categories of operators can be executed in different parts of a potential Virtual Machine. The Communications Units of a VM is the place for the data operators execution. The arithmetic operators can be optimised on the specific Execution Unit or the accelerator board of a VM. In fact, current general-purpose hardware developers provide most of matrix arithmetic functions as library routines. Intel 80860 and TMS320C40 Digital Signal Processors are such examples. The main difficulty is in the execution of neural operators. Currently hardware developers are working on the efficient implementation of these functions of parallel hardwares. As part of the Galatea

Project, Siemens and Philips are currently developing Virtual Machines for the efficient execution of these library functions. For the time being the neural operators can be executed sequentially on the local CPUs of VMs. This separation of neural operators, is the justification for the Virtual Machine philosophy of local CPU and parallel accelerator board combination.

In this thesis simulations of *MATLIB* programs are used for assessing mapping and execution strategies. Developing *MATLIB* simulations involve the following steps; programs are edited, using available library functions, then they are compiled by using standard *C* compilers, and finally run like any *C* program. As *MATLIB* is an open-system, new routines can be added easily, using the same parameter passing mechanism. When programming a new algorithm, if a specific matrix operation is not in the library, either the library is extended, or normal *C* functions can be used in conjunction with the *MATLIB* operators. *MATLIB* programs can also be ported onto different target machines by using cross-compilers generating hardware-specific code. In addition to eliminating parsing and interpreting, the use of *C* libraries also meant that debugging is a known process for the standard *C* programmer.

5.3.2. Parallelising *MATLIB*

A number of data communications operators have been included in *MATLIB* to enable parallel simulations on the Local Area Network at UCL. Standard TCP/IP sockets have been used to develop a number of simple functions that can be called in *C* programs with *MATLIB* functions. These functions are used to open a server, to connect to an open server, send integer or double precision values or matrices through the open server. Workstations connected to the Ethernet network can then be used, as parallel processors or as simulations of Virtual Machines.

The following two functions are developed to open a socket by the server, and to connect to an open socket by the clients;

- *opensocket* (socket_array_addr, total_VMs); This function opens a TCP/IP communications channel, waits for the number of clients defined in the *total_VM* and returns the socket addresses in the integer array socket_array. It is used by the server or the scheduler program. Each client can be then addressed using array reference, such as *vm[1]*, or *vm[2]* etc.

- *consocket* (socket_id, client_no); This function is used by the clients to connect to the already open socket. The socket_id is used to address the scheduler throughout the program. Parallel programming with *MATLIB* is simple.

A list of the communications operators are provided in Appendix B. These functions are blocking *get_* and *put_* statements as they block program flow until data are read or written respectively. *put_* statements don't block the execution until the buffer is full, but all *get_* statements are blocking they halt operation until the request is satisfied. In the use of the *get_* statements, these statements must be matched by *put_* statements at the other end.

In addition to the blocking and matching data exchange commands a passive server has also been developed which uses the following routine.

- *servis* (socket_id); This routine in a loop scans all open socket links and checks whether there are any data transfer requests. *servis* function needs a non-blocking read statement. A blocking read would result in the halting of all communications between the message passing modules.

Clients use the passive server with the help of a new data transmission command:

- *post* (socket_id, destination, matrix_name); This command uses the open socket to send data to a third party using the passive server. The passive server parses the *post* request and forwards data to the desired destination.

5.4. *NETLIB* Neural Network Library

The *MATLIB* library functions are extended to a parallel neural network library, *NETLIB*. This incremental approach of building high-level libraries from simple building blocks is a modular and plausible method. The library allows neural network functions to be called from *C* programs by setting up a number of parameters, without a detailed knowledge of neural network programming. Currently, *NETLIB* consists of the recall and training functions of the Hopfield, the SOM and the Backpropagation models.

hrecall (Inputs, Weights, Outputs, Iterations);

The Hopfield recall function requires Inputs and Weights matrix data structure pointers, and the maximum number of iterations allowed for convergence of a single pattern. The Outputs matrix contains the system's response.

```
slearn ( Inputs, Weights, Outputs, GridDimension,
           Iterations, DisStart, DisStep, DisEnd,
         GainStart, GainStep, GainEnd );
```

```
srecall ( Inputs, Weights, Outputs );
```

These two functions are designed for the learning and recall phases of the SOM. The *slearn* function requires the Inputs, the initial Weights matrices and a number of parameters for network training. These parameters are the dimensionality of the output grid, the maximum number of training iterations, the start, decreasing step and the end of the neighbourhood distance, the start, decreasing step and the end of the gain (or the learning rate). The trained weights are obtained in Weights matrix and the system response for given Inputs are returned in Outputs. The *srecall,* on the other hand requires only Inputs and Weights matrices, and return the Outputs. The topology of the network is hidden in the dimensions of the matrices involved in the recall procedure.

```
bplearn ( Inputs, Targets, Weights1, Weights2,
       Iterations, Gain, Momentum, Tolerance );
```

```
bprecall ( Inputs, Outputs, Weights1, Weights2 );
```

Tuned for a three layered Backpropagation topology, *bplearn* requires, Inputs, Targets matrices, initialised weight matrices (Weights1, Weights2), the maximum number of Iterations, the Gain (or the learning rate), the Momentum term, and the Tolerance level for error in learning. The weight matrices could be either randomised values for training or partially trained weights for retraining. After the training the network's response can be obtained by calling the *bprecall* function, which returns Outputs matrix.

As the algorithm library functions are built by using the *MATLIB* functions in the first place, the *MATLIB* functions can also exploit general-purpose parallel architectures. Complete networks can be mapped onto independent processors or VMs which are tuned to execute matrix-based arithmetic operations. Three levels of parallelism can be realised by using the *NETLIB* functions together with parallel *MATLIB* functions. Firstly, on the highest level, *NETLIB* functions can be executed in parallel by mapping complete networks onto parallel processors. Secondly, on the intermediate level, *MATLIB* functions within each *NETLIB* function can be parallelised. Finally, at the operation level each *MATLIB* function can be executed on a parallel systolic array of processors.

Exploiting the highest level parallelism, complete networks can be executed in parallel or data can be pipelined through a series of neural network algorithms. *NETLIB* allows various combinations of networks to be built and experimented on as hybrid neural networks. One major gain can be made by using network level parallelism: the problem of neural network design can be computationally solved. This is done by simulating a number of parallel networks initialised with different parameters, competing to solve the same problem. As a result of this evolutionary competition, optimum network design could be achieved. Similarly, parallel networks can be trained to specialise in different parts of a complex training dataset, thus reducing the training time.

5.5. Summary

In this chapter, a number of neural network representation techniques have been analysed with programming examples, and the strengths and weaknesses of each approach were discussed. Matrix-based *C* libraries have been put forward as clear, modular representations for neural networks, facilitating mapping and efficient execution on high performance, general-purpose, parallel platforms. *C* libraries *MATLIB* and *NETLIB* have been designed and developed and the three major models have been simulated using the functions of these libraries. The analysis and discussions in this chapter conclude that:

Function-oriented programming techniques represent algorithms as a sequential set of tasks or procedures. Their use is widespread in the test and design stage of algorithms. In this class, *C* is the most popular neural network programming language amongst programmers. The power of *C* stems from its availability and the popularity of its low-level features. Using these low-level features, it is possible to program in a function-oriented or object-oriented manner.

Object-oriented representations provide a conceptually plausible framework for neural network programming. Object-oriented programming languages are suitable as high-level languages providing a user-friendly environment for programming. Most high-level languages are based on the best known OOP language *C++*. A less pure approach is to use hierarchical data structures and to build a generic neural network tree structure, consisting of layers, clusters, neurons and synapses. In fact, *C++* provides a pre-processor which translates objects and classes of the language into *C* data structures. The *C* representation is then compiled and executed on a sequential execution environment. A new trend in this area is towards parallel object-oriented languages that allow users to control parallelism explicitly or implicitly. Object-oriented and data structure-based representations facilitate mapping neural networks onto neuron-based massively parallel architectures and special-purpose neurocomputers.

Vector-oriented representations capture properties common to most neural network algorithms and provide a suitable environment for executions on general-purpose neurocomputers and fast vector based parallel architectures. Galatea's *VML* is a good example, but it suffers from "yet another language" syndrome. *MATLIB* on the other hand is similar to *VML*, with the added advantage that it provides flexibility and openness as it consists of a set of *C* functions. *MATLIB* functions are studied in four categories (Appendix B). Data and communications operators can be implemented on current hardwares. Most matrix arithmetic operators are provided by parallel hardware suppliers, and the high performance execution of neural operators can be possible on programmable parallel processor arrays. The three neural network models which are the focus of this research, have been simulated using the *MATLIB* functions (Appendix C). The *MATLIB* functions are grouped together forming a high-level library, *NETLIB*, which facilitates neural network programming. Parallel features of *MATLIB* enable the testing of parallel mappings and simulations on a SUN LAN. Parallel simulation results from this environment, are presented in chapter 8.

The design and development of *MATLIB* meets some of the objectives set for this research work. *MATLIB* captures properties common to the three models, it is flexible and modular, and it promises a high performance execution, through automatic mapping on general-purpose parallel hardwares which is the main focus of research.

Chapter 6

Mapping Strategy

This chapter presents motivations and design considerations for a generic Mapper. Mapping techniques are reviewed, and computational optimisation is chosen as a general and flexible strategy for mapping neural networks onto parallel hardware. A Computational Analysis Tool is designed to detect parallelism in MATLIB representations, an Automatic Parallel Mapper is put forward to automate mapping.

6.1. Design Considerations

In the wake of proliferating neural network models and applications, current sequential hardware platforms are not able to match the increasing requirements for faster processors and larger storage capacities. Using parallelism, general-purpose and cost-effective execution of neural networks can be achieved. A major difficulty in this is the parallel programming or mapping onto parallel hardware. Ultimately, what is desired is a mapper with a generic mapping strategy which is capable of automatic generation of parallel code from sequential representations. The following considerations are important in the design and implementation of such a mapper.

High Performance - Achieving high performance is the first objective in the design of a mapper. The high performance requirements vary in the training and recall phases of neural network simulations, depending on the data size, the model and the problem domain.

In training, fast convergence is required, especially in the research and development area, where a series of tests is carried out to establish, verify or benchmark a model or an application. Financial forecasting applications involving real-time training are such applications, as networks are expected to adapt and respond appropriately. For example, financial forecasting simulations with the Backpropagation model, reported in chapter 4, require the testing of many network configurations and parameters until acceptable results are obtained. This is partly because of the difficulties in setting up the correct parameters for the Backpropagation model, and partly because the Backpropagation model is notoriously slow in training. Indeed, most neural network simulations are computationally demanding during training due to the differential equations involved in their weight update routines. These equations are often expressed as difference equations in computer simulations which require repetitive operations. Fast processors are needed to satisfy this computational demand. The high level connectivity of neural network architectures are often simulated on conventional computers by using Random Access Memories. Large and high speed RAMs are necessary to improve the performance of simulations on these machines [6]. In addition to these requirements, simulations on distributed parallel systems require high-speed, wide-band interprocessor communications links.

In recall, real-world problems often require a real-time response. This is a level of performance which is acceptable to humans. A pattern is presented to a trained network, a single pass execution of the network takes place and a quick response is expected. When there are many patterns, the accumulative performance is also needs to be quick. Vision and speech applications are particularly demanding in recall, as networks must respond in very short periods. A neural network speech recognition system has to correctly recognise speech within milliseconds, otherwise the solution would be worthless. Vision problems involve the processing of large grids of data matrices as patterns for neural networks. Another demanding example is the real-world implementation of the Hopfield pattern recognition application, reported in chapter 4. The telecommunications system implementation of this application requires convergence rates in recall, in the level of milliseconds or microseconds.

Generality - The second consideration is the generality of the mapping approach. A proposed mapping strategy must be applicable for a range of neural network models, applications and parallel hardware platforms. The strategy should also be easily extended to other neural network models and applications, and onto heterogeneous hardware platforms. If the strategy is general enough, it could be applied to fine-grained non-neural network applications such as graphics-based problems.

Parallelism - High performance must be achieved through the efficient exploitation of parallel resources. Three types of parallelism which are applicable to neural networks are considered; *pattern parallelism*, *data parallelism* and *task parallelism*.

Pattern parallelism, also called training parallelism, is a way of speeding up the training process when large datasets are involved. Large datasets are divided and distributed over a number of parallel processors, which run identical neural network simulations. This method is applicable to a number of models which can carry out batch weight updates, during training. With the batch updates method, the weight changes are accumulated until all patterns are presented, then the changes are averaged and the update takes place. In a parallel system, each network can be trained on a subset of data, on an independent processor which communicates only the weight changes with a central processor, at regular intervals. As a result, a 100% parallel load balance can be achieved with little communications requirements. But this method cannot be applied to all models as it relies on batch weight updates, and this form of parallelism cannot be used in real-time recall operations.

Data parallelism involves the partitioning and distribution of the neural network structure. Structural partitioning examples shown in chapter 4 are examples of this type of parallelism. Object-oriented representations and fine-grained parallel hardware platforms are more suitable for data parallelism, where objects or groups of objects from the representation domain can be mapped onto message-passing, parallel distributed processors.

Task parallelism, on the other hand, is the natural consequence of using function-oriented representations. These representations can be parallelised by detecting concurrent, independent task paths and generating task parallel or pipelined code. Due to the data dependencies between tasks, this type of parallelism is more suitable for coarse-grained parallel architectures, where groups of tasks can be pipelined through the parallel processors. Although, it can be useful in training, its application to the recall phases is limited. Data parallelism and task parallelism are examined further in this chapter and communications costs for these types of parallelisms are parameterised.

Scalability - Increasing the number of parallel processors in the system should result in a faster execution.

Flexibility - Mapping and execution strategies must be flexible for update, modification, future expansion and manipulation by the user.

Modularity - Another consideration is the modularity of the mapping/execution strategy. Applications which require multiple neural or hybrid solutions are recently on the increase. A modular design would allow the interfacing of multiple networks, genetic algorithms and expert systems in a framework as independent and integrated modules. Modularity is necessary both in software and hardware environments, and the mapping strategy must adopt and exploit this.

Automation - The final consideration is to achieve the parallel mapping automatically. This is particularly challenging considering the number of models, software representations and hardware platforms. Various degrees of automation can be considered, such as *manual*, *semi-automatic* or user directed mappings which may be necessary in some cases, but a fully automatic mapping, resulting in high performance parallel execution is the ultimate target.

6.2. Mapping Techniques

A number of research groups have mapped popular neural network models onto general-purpose parallel processors. These hardware platforms include massively parallel processor arrays, one and two dimensional systolic arrays, Digital Signal Processors (DSP), Transputer arrays, and hypercube processor architectures. Most of this previous work involves mapping specific models onto specific hardwares aiming for high performance. The Backpropagation model is the most popular model and has been simulated on a number of parallel hardware platforms.

The Backpropagation model has been simulated on an SIMD architecture, the Connection Machine, and later, on its improved version CM-2 [155]. Simulations on CM-2 make use of the 2 dimensional, nearest neighbour communication link facility provided by the system. Similarly Watanabe et al. reported an implementation of Backpropagation on the massively parallel cellular array processor AAP-2 [149], which is also a 2 dimensional, 256x256, mesh-connected array of processors. The performance of AAP-2 on the Backpropagation model reaches 18 MCPS (Million Connections Per Second). Both techniques distribute the Backpropagation neurons onto SIMD processors are examples of structural or data parallel mappings. Other SIMD examples are the implementation of the Backpropagation model on the CNAPS Neurocomputer chip [109] and HNC's work on linear floating point array SNAP machine [111].

One of the early reports of successful mapping of the Backpropagation model onto a MIMD processor array, came from Carnegie Mellon University [120]. This work reported the mapping of a Backpropagation network onto a linear, 10-processor array Warp machine. The results show the simulator is able to perform at 17 Million CPS. Two different mapping techniques have been used on the Warp machine; the first one involves the partitioning or spatially mapping of the Backpropagation network structure, and the second technique involves pattern parallelism, which reduces the training time considerably. In this case, the dataset is divided into 9 subsets and distributed over 9 Warp processors. Each one of the 9 processors runs identical Backpropagation networks, while the 10th processor updated the global weights and pumped the new weights through the network.

An interesting example of the parallel mapping of the Backpropagation model is the distribution of the network over the Intel iPSC/860 hypercube [79]. The network partitioning (data parallelism) technique is applied onto the MIMD architecture of the Intel machine. An increase in the number of processors results in a higher performance reaching 11 MCPS with 32 processor. This work also reports complications in data parallel mapping of Backpropagation networks with odd number of neurons, because of the difficulties in achieving a load balance. Another example of mapping on the Intel iPSC/2 machine focuses on the decomposition of networks based on a computational load analysis [56].

A number of research groups have mapped Backpropagation networks onto general-purpose, high performance, digital signal processor (DSP) arrays. Iwata et al. report an implementation of the Backpropagation on a 4 DSP ring coupled architecture, Neuro Turbo [78], which is an neural network accelerator board for the NEC personal computer series PC98. The mapping technique focuses on the mathematical equations involved in a Backpropagation simulation. The equations are distributed over the 4 processors and executed in parallel. The 4 processor system produces 2 MCPS, but the system is scalable and produces $1.8n$ MCPS for each one of the n processors in the system.

The self-organising feature maps have also been mapped onto parallel hardware. One of these mappings is an example of pattern parallelism [103] where a training dataset is divided into 10 processors of the Warp Systolic computer. This work adopts the batch-weight update method for the SOM to reduce the interprocessor communications load. It is also reported that this approach sometimes produces unexpected results, such as maps folding, instead of an orderly distribution of input vectors in the output grid. Another example of mapping the SOM is an attempt to parameterise the mapping process [68]. It involves structural partitionings similar to the mappings suggested in chapter 4. One-dimensional and two-dimensional processor arrays are explored, and the work shows that the system scales for large networks.

Transputers have been used in a number of topologies as parallel hardware platforms for neural networks [44, 112, 113, 154]. The SOM has been mapped onto Transputer arrays [136]. This work makes use of 16 Transputers in a ring topology; the system is said to be scalable, as the number of Transputers increases, the system performance increases.

Hopfield nets also have been mapped onto Transputer-based neurocomputers [83, 138]. These simulations use structural network partitioning techniques and distribute nodes of the output grid across a number of processor arrays.

Conclusion

These techniques are model-, hardware- and sometimes application-specific. Some generic approaches have been suggested [94, 95], but they do not address generic representation issues. Most of the techniques achieve a high-performance by partitioning neural network structures, or dividing training patterns to identical neural networks running on a number of parallel processors. Sometimes these mapping techniques modify algorithms, so interprocessor communications are reduced. These techniques are not general, flexible, nor can they be automated. Following the neural network analysis in chapter 4, and the assessment of representation techniques in chapter 5, this thesis has established a matrix-based generic representation for a range of neural network models and applications. This generic representation is now supported by a generic mapping strategy.

6.3. The Strategy

The mapping strategy cannot be seen in isolation, as it is closely linked to the execution and the representation strategies.

Execution Strategy - In chapter 2, two possible paths for neural network execution were presented; special-purpose and general-purpose neurocomputers. Special-purpose neurocomputers often emulate the neural features on hardware, aiming for high performance for specific models and applications. General-purpose neurocomputers, on the other hand, provide a high performance execution for the computations involved for a wide range of neural network models and applications. The general-purpose execution philosophy is more general, flexible and cost-effective.

The execution strategy adopted in this research is based on exploiting general-purpose parallel processors in the execution of neural network computations. The overall system is a GPNC, with a comprehensive programming environment and a number of independent, high performance parallel processors. These general-purpose modules contain local memory and accelerator board-based processors, which communicate through a message passing protocol. A bus communications architecture is presumed as it would provide the most flexible interprocessor communications scheme. The system scales well; ie when the number of processors are increased, the system performance also increases.

Representation Strategy - The neural network analysis in chapter 4 shows that the most common operations in neural network simulations are computationally intensive, repetitive multiplication and addition operations, with some neural-specific functions. Most of these operations could be abstracted in a vector-oriented representation. In chapter 5, a matrix-based library was designed to show that most neural networks can be represented using simple vector/matrix arithmetic operations. General-purpose neurocomputers provide high performance execution platforms for these vector/matrix operations. Using a number of parallel general-purpose devices to increase performance is a cost-effective and scalable approach. This solution requires efficient mapping strategies to partition and distribute neural network representations across a number of parallel modules. The main duty of a mapper is to exploit these increasingly parallel environments. Matrix-based representations have been chosen as they are able to capture computationally intensive neural and non-neural characteristics, and they are suitable for general-purpose parallel hardware platforms.

Mapping Strategy - The mapping strategy is to design a mapper as a computational optimiser process aiming for a high performance, general, flexible and efficient execution. Although the strategy is global, the mapping efforts in this thesis focuses on the optimisation of the use of a small number of powerful parallel computers, as modules of a general purpose neural computer. This mapping strategy has a number of advantages. The main advantage is that it is the most general approach, ie the strategy can be applied to other problems with little change. Secondly, once the problem is defined as an optimisation problem, a number of optimising approaches can be applied. These solutions range from the straightforward computational cost analysis to the use of neural networks or genetic algorithms. Parallelism introduces a new computational cost; communications costs. The optimiser mapper aims for a high performance execution by minimising:

1 - Processing costs through achieving a load balance.
2 - Communications costs.

The mapper achieves its task by projecting many possible partitionings of the neural network representation, deciding on an optimum partitioning and distributing the representation onto parallel hardware. The projection operation involves the calculation of potential processing and communications costs for the possible parallel mappings and execution.

This chain of execution, representation and mapping strategies can be viewed in the context of a General Purpose Neural Computer. This framework contains a coarse number of Virtual Machines, and each VM contains a local CPU and a fine-grain parallel processor array. Parallel mapping is necessary in both levels: *(i)* at the high level, neural network representations are mapped onto a number of VMs, and *(ii)* at the low level, within the VMs, matrix-based instructions are mapped onto a parallel processor array.

The mapping strategy developed in this thesis aims to partition neural network representations to a coarse number of parallel VMs. Although similar requirements are valid within the VMs, this level of mapping is a hardware design issue, as it involves the optimisation of the matrix-based operations on parallel processor arrays with specific memory and communications characteristics.

Parameters of the Mapper

A number of parameters play an important role in the mapping decision for parallelism. These parameters are application, model, representation and hardware-related.

- Application-related parameters are directly linked to the problem domain. Some problems require single domain neural networks, and others can be solved on multi-domain, modular or hybrid systems. As a result, parallelism may be required at the highest level; the application or the network level. Cooperating or competing networks are examples where parallelism is required at this level.

- Model-related parameters are hidden in the *connectivity*, *functionality* and *sequentiality* of neural network algorithms.

Connectivity - of a neural network can be treated separately from the functionality of the network. It is the definition of rules relating to operations such as copying or modifying data, between the different parts of the network. For most algorithms, the role of connections is a simple copy operation, if the multiplication by weight operation is included as part of the processing element's functions. In *pruning* and *growing* networks, the connectivity is treated as an active object with a capacity for expansion and self-modification. This approach also allows connectivity to be more complex than a copy operation. In the case of matrix-based representations, the connectivity information is expressed in the dimensionality of the matrices that describe the network topology, and these matrices are treated like any other data matrix.

Functionality - describes the operations involved within the processing elements or the nodes of a neural network. These operations may be simple scalar arithmetic operations or complex functions depending on the complexity of the algorithm. In hierarchical or object-oriented representations, functionality can be described in various levels of granularity. Matrix-based representations focus on the overall functionality of the algorithms, rather than the fine-grained description of the processing elements.

Sequentiality - defines the temporal relationship between different functional modules. This relationship is often implicitly expressed in the order of the instructions in computer simulations. In *C*, *VML* and *MATLIB* representations the program flow indicates the sequence of instructions. In contrast to sequentiality, parallelism can be instructed either explicitly, or sequential programs can be analysed, concurrent paths are detected and parallel sections of code can be generated.

- Hardware related parameters comprise; *granularity*, *communications scheme*, and *processor speed and memory*.

Granularity - describes the hardware platforms with the number of processor they contain. On the one extreme, there are massively parallel architectures comprising hundreds of simple processors, on the other, there are coarse-grained or single processor architectures.

Communications Scheme - is possibly the most important parameter in the parallel mapping. The interconnection topology of the parallel processors, communications bandwidth and speed must be evaluated in parallel mapping. These characteristics can be obtained from the manufacturers.

Processor Speed and Memory - directly determine the efficiency of the execution. On a parallel architecture, the processors' performance, local memory capacity, and memory access times must be considered. These features are also provided by the manufacturers.

6.4. Cost of Parallelism

Data and task parallelisms, are discussed below in terms of the communications costs they introduce, in an attempt to parameterise these costs and automate mapping decisions.

Data Parallelism

A data partitioning example, to describe the communications costs, is as follows. Let us consider a matrix operation involving the element by element multiplication of two matrices, executed on sequential hardware. This operation has fine-grained components such as elements of the first matrix, and the elements of the second matrix that are multiplied. The matrix operation can be separated into subcomponents involving a number of multiplications of the submatrices down to the element level, which would involve the multiplication of two scalars. The submatrix element multiplication operations can be distributed and executed simultaneously on a number of parallel processors, as they don't have data dependency. The results can be reassembled producing a single result matrix, which would be identical to the result obtained by the sequential execution. If the operation can be evenly chopped and distributed, a balanced load on similar parallel processors would take an equal amount of time. The distribution of the subcomponents and the reassembling of the results would be the communications costs on the parallel execution (Figure 6.1).

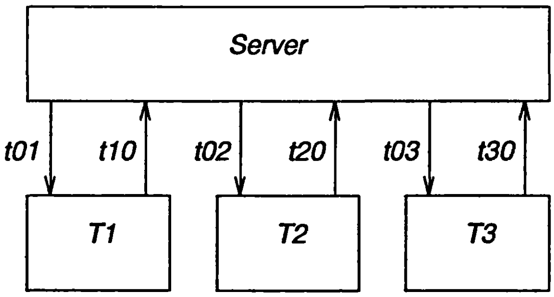


Figure 6.1. Communications Costs in Data Parallelism

To parameterise the problem, consider that a matrix operation takes T seconds on a sequential computer, and the distribution of the submatrices onto 3 identical parallel processors takes $t_{01,}$ $t_{02,}$ and t_{03} seconds, respectively (Figure 6.1). Assuming divisibility by 3,. a parallel execution on similar processors would take $T/3$. Finally, reassembly costs for the results are $t_{10,}$ $t_{20,}$ and t_{30}. The total computational cost for the parallel execution can be estimated as;

$$T_p = t_{01} + t_{02} + t_{03} + (T/3 - t_{02} - t_{03}) + t_{10} + t_{20} + t_{30}$$

Depending on the total cost, a decision can be made to partition or not to partition the instruction into parallel components. Although a sequential server is assumed in this example, a concurrent server would present a similar problem. In that case, the simultaneous transmission of the submatrix data from the host to/from the parallel processors must be considered, and these would be constrained by the bandwidth limits.

Task Parallelism - Pipelining

Instruction pipelining is a way of speeding up repetitive operations. It is used extensively on recent high performance hardware platforms such as RISC architectures. A sequential program can be pipelined through a linear array of processors. A pipeline can be organised on systolic array processors with one instruction per processor, or data-independent sections of code can be identified and mapped onto each processor of the array. There are two considerations for setting up a pipeline:

1 - The section of code must have a repetitive set of operations.

2 - The section must have one-directional data-flow.

Suppose that a section of code containing 3 instructions, is repeated N times, and it has only a forward data-flow. Assume that each instruction on the sequential host takes T_1, T_2, and T_3 *seconds*. Repeating the execution N times takes:

$$N * (T_1 + T_2 + T_3) \ \textit{seconds}$$

Let the communications costs for the transmission of data and arguments between the 3 processors and the host, be $t_{01,}$ $t_{12,}$ t_{23} and t_{30} (Figure 6.2). Assuming all processors and the host are identical and processing costs are bigger than communication costs, the pipelined execution would last:

$$T_p = t_{01} + T_1 + t_{12} + T_2 + t_{23} + T_3 + t_{30} + (N-1) * \max (T_1, T_2, T_3)$$

The partitioning/mapping decision can be based on this calculation. An even load balance, coupled by minimised communications requirements can result in reduced execution times, especially when a high repetition rate (N) is involved. Loops with reverse flow data paths cannot be pipelined, as the beginning of the loop requires data from the end, and this would block the data flow in the pipeline.

Both types of parallelism can be guided by explicit instructions from the programmer who knows a near optimum partitioning. An example of this is, the *nC* PAR statement which instructs the compiler that the following operation can be executed in parallel. Alternatively, in task mapping, vectorisers, compilers or optimisers can be used for the automatic generation of parallel code from sequential representations. Fully

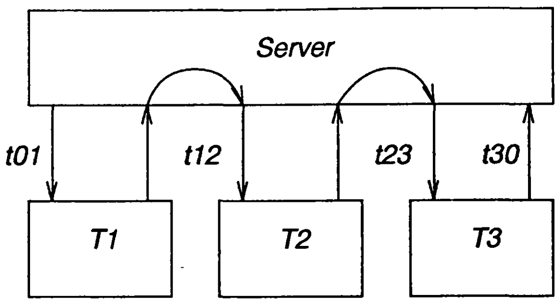


Figure 6.2. Communications Costs in Task Parallelism

automatic mapping is one of the objectives of this thesis. Based on the cost analysis calculations outlined above, a Computational Analysis Tool and an Automatic Parallel Mapper are put forward to achieve automatic mapping.

6.5. Computational Analysis Tool

As the basis for parallel mapping is the computational execution cost, a computational analysis tool (CAT) has been developed to calculate all aspects of processing and communications costs. Using this tool, a line by line, and overall cost profile is generated without executing the code. The profile can be used as the basis for data or code partitioning for data parallelism or instruction pipelining.

To profile *MATLIB* programs, first, a computational costs lookup table is set up by executing *MATLIB* functions on the potential hardware, and the processing cost for each call is modelled. After this, CAT can estimate the computational cost for any algorithm as long as the algorithms are written, using *MATLIB* in a restricted syntax format. These restrictions are applied to ease the parsing, and to be able to calculate computational costs during compilation time. Below are the list of these constraints on the *MATLIB* definitions which must be followed for processing by CAT:

1 - Programs must be written by using *MATLIB* functions. If new matrix operators are added to the library, their estimated unit computational costs on the potential hardware must be added to the computational look-up table.

2 - Programs must be written in single *C* main function listings, with no rule hierarchy. Data definitions and execution statements must be kept on top of the the same file.

3 - Each *MATLIB* (or *C*) statement must be typed on a separate line. This is a requirement to ease the parsing operation, to be able to generate a line-by-line computational profile

and to allow a line-by-line code partitioning for task parallelism.

4 - Only 'for' loops with constant loop indices are allowed. These constants can either be defined at the top of the file as 'int', or scalar values can be used directly. This enables the calculation of the number of repetitions involved in each loop. In fact, this is not too restricting, as in practice, most of the loop indices relate to the network topology which is defined at the beginning of the execution. Often, the only unknown loop parameter is the number of iterations to reach the convergence. This is not a big problem, as a large enough number of iterations can be assigned at the beginning and the loop can be broken if the convergence is reached, earlier than estimated.

5 - Only 'if' control statements are allowed. This is often sufficient for checking convergence and quitting the execution when convergence is reached.

6 - 'for' loops and 'if' control statements must be opened with '{', and ended with '}' operators which must be typed on a separate line.

CAT is a simple simulation tool which calculates computational costs based on given criteria. It estimates sequential, data parallel or task parallel costs, following the cost calculations outlined in section 6.3. CAT has the following functional modules.

- *MATLIB Parser*
  The parser is a *C* program module, which scans a *MATLIB* program, identifies constants, matrices, loops and *MATLIB* functions. Only matrix operations are examined, as these are the main bulk of neural network computations and communications. The parser equipped with the *MATLIB* calling protocol, identifies which variables are read and which are written at each program line. As a convention, the first arguments are always the destination and these variables are registered as written by the calling function. The records are used in the variable analysis to generate data dependency paths which are necessary for partitioning. The 'for' loops are parsed and registered into loop data structures with the starting line, the ending line and the number of iterations involved. The parsing of the 'for' loops require the parsing of the 'if' statements as they share the common closing operator '}'.

- *Loop Analysis*
  Loops are essential for neural network programs as all neural network programs contain repetitive operations. The number of repetitions in any line of the program can be calculated by parsing the loops, and this repetition value can be used as a

coefficient to calculate the computational cost for the line and the total cost for the program. The same repetition value is also used to calculate communications costs in the case of splitting the data or the program in a specific line. But the loops have a much more important task than this; they provide reverse data flow within the program by carrying data values from later stages to the previous lines. This is the case when a variable is first read within a loop, then written somewhere else within the same loop. This will be further explained in the variable analysis.

- *Variable Analysis*

  As *MATLIB* programs are listings of *C* statements, any partitioning technique must identify the concurrent and independent data flow paths. Normally, a program with no loop statements would be a series of data write operations followed by a read, further read and write operations. Loops operations complicate this straightforward data flow. The variable analysis routine is responsible for the generation of data paths which are dumped for analysis by the programmer. These paths are labelled as forward paths which show variables which emerge at certain lines of the code as a result of a write operation; they are read, or read and written in the following lines. Reverse or backward data paths involve variables read and written at the early stages of a loop, then rewritten in the same loop. So the loop structure carries the new data to the beginning of the loop. Forward or backward, the data dependency is very important if program or data partitioning and parallelism are considered.

The Computational Analysis Tool provides the following information about the potential execution, without running the code.

1 - Total matrix memory usage in Bytes.

2 - Total sequential processing costs in seconds.

3 - Line by line listing of the computational cost and possible computational cost in the case of data partitioning and parallelism for a given parallel configuration.

4 - Line by line listing of the communications cost in the case of code partitioning in that line for a pipeline on a given configuration.

5 - Forward and backward data flow charts, which is list of symbols, for all matrices, indicating whether the matrix is read or written, is in a forward or reverse flowing data stream or if it flows in both directions, for each line of the program.

To use CAT, a *MATLIB* listing of the program is presented to the program 'map'. The computational analysis results can be used either as a guide for manual network partitioning and mapping, or in automating the parallel mapping process.

6.6. Automatic Parallel Mapper

The Computational Analysis Tool is extended to generate parallel or pipelined *MATLIB* definitions automatically. The result is the Automatic Parallel Mapper, which is a *C* function that exploits the variable and loop analyses carried out by CAT, and automatically generates parallel *MATLIB* definitions with complete data and function definitions, and matching data transfer instructions. APM mapping decisions are based on the same parameters provided to CAT. These are the number of parallel processors, their computational characteristics (in the computational lookup table), and the main parameter of the communications architecture; the transmission rate in Bits/sec. CAT analyses *MATLIB* programs and explores data and task parallelisms, depending on its projections and decision APM generates parallel code (Figure 6.3).

Data parallelism can be performed by exploiting the line by line computational analysis carried out by CAT. The pseudocode for the automatic generation of data parallel programs is as follows:

1 - Read the number of parallel processors, the communications speed, and the look-up table relating to the unit execution costs of the *MATLIB* functions on the hardwares involved.

2 - Calculate the sequential computational cost of the *MATLIB* operation for the host.

3 - Calculate the potential parallel computational cost for the current *MATLIB* operation on the available parallel hardware. This parallel cost includes the distribution costs, parallel processing costs and re-assembly costs.

4 - Compare the sequential computational cost with the parallel cost, and if the parallel cost is smaller, then generate parallel code.

The automatic generation of the parallel code involves the following:

- On the host side: the division of matrix data into the number of processors, the generation of *put_* instructions to transmit partitioned data, and *get_* instructions to reassemble the resultant matrix.

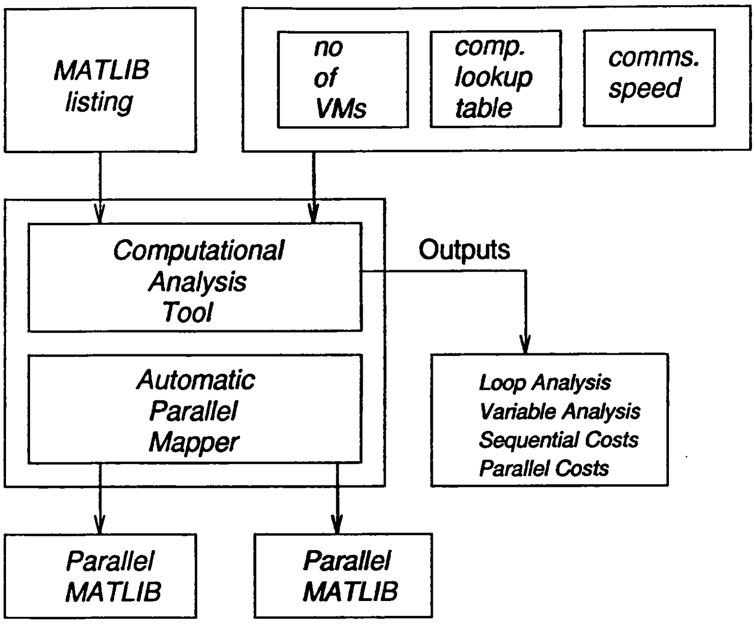


Figure 6.3. Automatic Parallel Mapper

- On the parallel clients side: the data definition statements, the matching *get_* instructions to obtained partitioned matrices, the matrix operation, and the transmission of results by *put_* which is matched at the host end.

Task parallelism is also possible using the CAT results. Repetitive sequential operations such as the stages of a network training can be pipelined through an array of processors. Certain heuristic rules can be followed to prevent partitionings which would result in a high interprocess communications. For example, loops represent a high interconnectivity, and they should not be broken unless there are subloops which can be mapped onto separate processors. The pseudocode for instruction pipelining is as follows:

1 - Read the number of parallel processors, the communications speed and the lookup table.

2 - Calculate the line by line, and total computational cost for the sequential execution on the host.

3 - Divide the total cost by the number of processors (assuming they are identical processors). This gives the ideal cutting lines with a perfect load balance for a pipeline.

4 - Cut only the forward data flow streams, and avoid to cut any backward flow data paths.

5 - Generate the host and slave sections of the code, with correct data definition and data exchange statements.

Once parallel code listings are generated, they are compiled using a *C* compiler and they are ready for execution the parallel system.

6.7. Summary

In this chapter, firstly, the motivations and design considerations have been outlined, other mapping techniques have been reviewed. A number of model-specific and hardware-specific techniques have been assessed. It was concluded that they cannot be generalised to all models and hardwares. The execution, representation and mapping strategies have been established. The execution strategy is based on the exploitation of general-purpose parallel processors. The Virtual Machine philosophy promotes the use of vector-based hardwares in a general-purpose neural computing framework. Matrix-based representations seem to capture neural network properties, and are suitable for general-purpose parallel hardwares. The mapping strategy has been outlined as the computational optimisation of execution costs, as a general, flexible and potentially upgradable approach.

The main challenge in the computational optimisation is the parameterisation and costing of the executions. For this purpose, the computational costs for data and task parallelism are parameterised. A Computational Analysis Tool has been designed and developed for analysing and profiling matrix-based *MATLIB* representations. Once the computational costing is achieved, the automatic generation of the parallel code is a relatively simple task. An Automatic Parallel Mapper has been designed for this purpose which exploits CAT results in the automatic code generation. Chapter 8, presents the CAT/APM projections and mappings of *MATLIB* neural network representations.

Chapter 7

Galatea Mapper and Scheduler

In this chapter, the Galatea Mapper and Scheduler are presented. As part of this thesis work, the Galatea Mapper is implemented, the Scheduler is specified, and using the GPNC simulator a number of parallel simulations have been carried out to test mapping strategies.

7.1. Introduction

This chapter presents the Galatea Mapper development as part of the Galatea GPNC simulator. Research presented in this chapter is the work of the author, carried out as part of the Galatea Project. The Galatea Mapper shares the same design considerations set out for the Mapper in chapter 6, and the mapping and scheduling strategies developed in this chapter are directly related to the research goals outlined for this thesis. Both the Mapper and the Galatea Mapper are designed to map matrix-based representations (*MATLIB* and *VML*) onto coarse-grain parallel architectures. This chapter presents the Galatea Mapper in the context of the Galatea GPNC to highlight the real-world constraints and the outcome.

The Galatea General Purpose Neural Computer is an advanced architecture, designed for the execution of a range of neural network models and applications. In chapter 3, a detailed description of the Galatea system is provided, including its general programming environment and multi-processor parallel architecture. In this chapter, the Galatea Mapper is presented in the context of the Galatea GPNC automatic code generation chain. As the mapper is responsible for the initial scheduling of the parallel execution, the scheduler development is also described.

As seen in chapter 3, the Galatea GPNC is a coarse-grained parallel architecture, which brings together generic hardware devices called Virtual Machines. Each VM consists of a *Communications Unit*, responsible for the control of the accelerator board, and an *Execution Unit* containing an accelerator board. The communications unit typically consists of a CPU and a large local RAM. Execution units are based on fast matrix operator generic boards, currently being developed by Siemens and Philips [41,42]. VMs communicate and interpret *VML*, a matrix-based, intermediate-level language, consisting of scalar and matrix arithmetic operations. Consistent with the VM philosophy, the user-interface is designed in a parallel distributed fashion with a number of independent processes called Graphical Virtual Machines (GVM). All the independent modules of the GPNC; VMs, GVMs and the Scheduler, are interfaced with a message passing communications protocol.

The Galatea Mapper's duty in this environment is to partition and distribute the *VML* representation across a number of parallel VMs. The Mapper is responsible for the initial scheduling of the execution, and for generating necessary data transfer instructions accordingly. The Galatea Mapper processes raw *VML* and generates parallel *VML*. This is essentially *VML* with a number of data exchange commands, which are interpreted by the Scheduler and the Communications Units of the VMs. After the initial mapping, the execution is controlled by the Scheduler, which serves as the communications interface.

In this chapter, first, the Mapper is defined in the context of the Galatea GPNC automatic code generation process. Secondly, the Scheduler, the communications protocol and the run-time operations are explained. Then, the implementation steps, the practical issues relating to the implementation and problems encountered during the implementation, are discussed. Finally, Galatea mapping examples and parallel simulations on the Galatea GPNC are presented.

7.2. Galatea Mapper

The Galatea Mapper is a building block of the Galatea General Purpose Neural Computer. It plays an important role in the correct partitioning, placement of the code, and scheduling of the parallel execution on the GPNC. The Galatea Neurocomputing group at UCL has developed a complete simulation of the GPNC, which runs on SUN workstations. As part of the GPNC simulator, an interpreter [140] is used to execute *VML* programs with a number of applications, including pattern recognition, image processing and financial forecasting. The GPNC simulator provides an adequate environment for testing various mapping and scheduling strategies. The mapper is an independent module in the GPNC simulator's automatic code generation chain.

The code generation process (Figure 7.1) for the GPNC involves the following stages. First, a user writes high-level language *N* code for the application. An algorithm library can be used to code applications in *N* (an object-oriented language based on *C++*). The Systems Architecture Builder (SAB) can be used during this process. Later, the *N* to *VML* compiler automatically generates the raw *VML* code. The compilation process involves vectorising and optimising the object-oriented representation to generate the *VML* code. During the compilation process a symbol table is also generated. The symbol table contains a data/rule correspondence table for cross reference purposes. The table maps names and entities between *N* and *VML*. The mapper processes the raw *VML* code following any existing user directives, and generates parallel *VML* code for specific VMs. The code also includes appropriate data movement instructions.

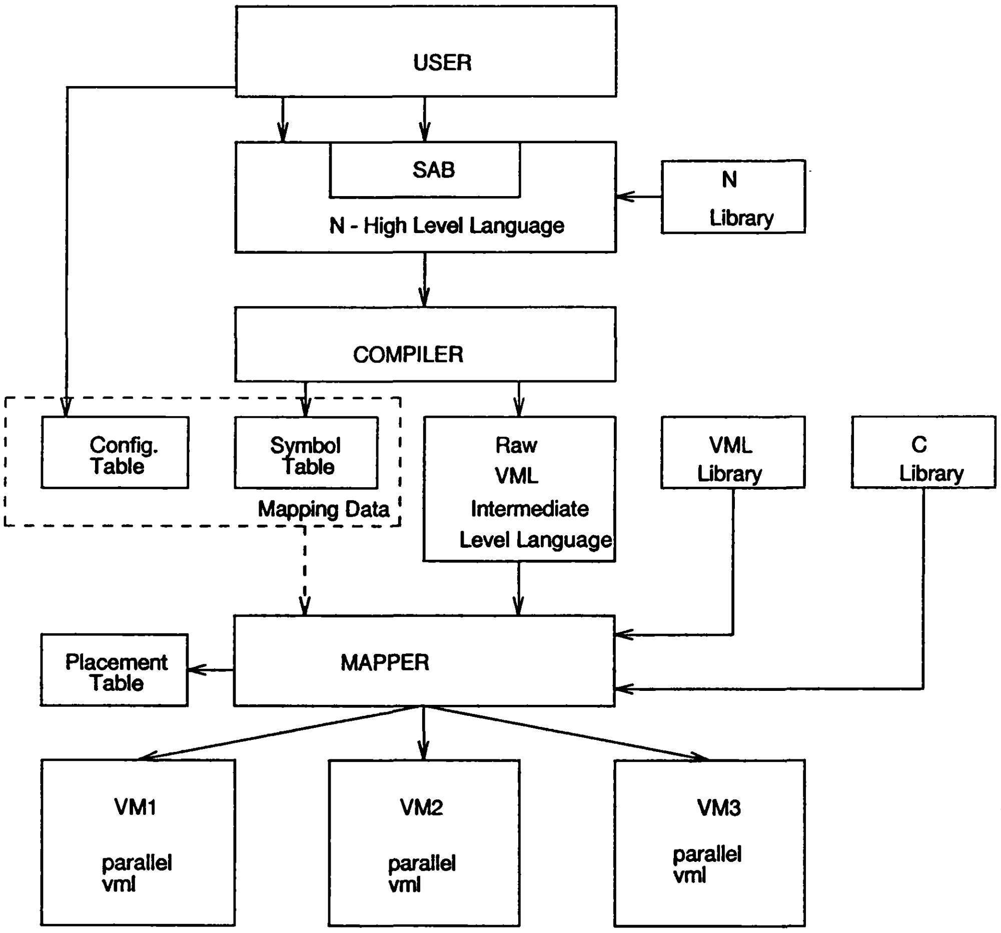


Figure 7.1. The Galatea Mapper

Once the *VML* representation is generated, the hierarchical object structures created in the *N* environment are lost. Only matrix and scalar names are relevant at the *VML* level. A user enquiry relating to a user-defined object name, has to go through a symbol table or a name generation scheme to find a correspondence in the intermediate-level matrix representation.

It is planned that a *C* library should be used to support operations which are not available in *VML*. These are sophisticated functions such as Fourier Transformations.

These routines can be called in a similar way as VMs using the GPNC's message passing communications scheme. A Placement Data file is also generated by the mapper. This tells the scheduler which *VML* rules, variables and data reside in which VM.

In the VM level, the Communications Unit CPU stores the *VML* code in its local memory. It parses and interprets the *VML* definition to fire low level *VML* commands. These instructions are similar to *VML* commands, but they are without control and loop statements. The communications unit generates scalar and matrix arithmetic operations. The scalar arithmetic operations are executed on the local CPU efficiently. The low level matrix operations are sent to the accelerator board, which in turn converts them to the microcode, for execution on the processor array.

The Galatea GPNC with its SUN workstation host and a number of Virtual Machines presents a coarse-grained parallel mapping problem. The mapping operation involves parallelising the raw *VML* representation and generating parallel *VML* for specific VMs with appropriate data movement instructions. *VML* listings contain a number of rule bodies and a main rule. In this form, task mapping is the primary mapping option and data partitioning is not favoured. Galatea believes that the VMs are powerful modules, and that they can meet both the memory and the CPU requirements for large applications. Typical applications for the GPNC would be multi-network applications with whole networks mapped on single VMs. The Mapper's task is then to partition the multi-network *VML* specification, and to achieve a parallel efficient execution. When single networks are involved, the mapper can map *VML* rules or tasks onto a number of parallel VMs.

Galatea considered two types of mappings depending on the time of the mapping. *Static Mapping* is the initial process that occurs during the compilation time, before the execution starts. *Dynamic Mapping*, on the other hand, is done during the execution, after re-assessing the results of the previous mapping. For example, if the load balance is not satisfactory, the execution can be frozen and the application is saved and remapped. Dynamic mapping can also be performed as a result of user instructions for modifying the configuration during execution.

7.3. Galatea Scheduler

Once the initial mapping is completed the execution can start. The Scheduler provides the run-time communications interface between the various modules of the GPNC. The Scheduler is a passive process, waiting for requests from users through the

Graphical Virtual Machines (GVM), and the VMs, carrying out instructions and supporting routine monitoring tasks. The scheduler is planned to be a single process running on the host machine, but it can also run on a Transputer based computer or distributed over a number of parallel processors. The duties of the scheduler are:

1 - Setting up and initialising VMs
2 - Transmitting data and rules to the VMs
3 - Analysing user requests and taking necessary measures
4 - Updating data values on the VMs
5 - Carrying out user monitoring requests
6 - Data exchange between VMs

A communication diagram based on the scheduler is shown in Figure 7.2 All physical blocks have message queues, and these queues are processed by routines which sort the messages with respect to their priority and arrival time.

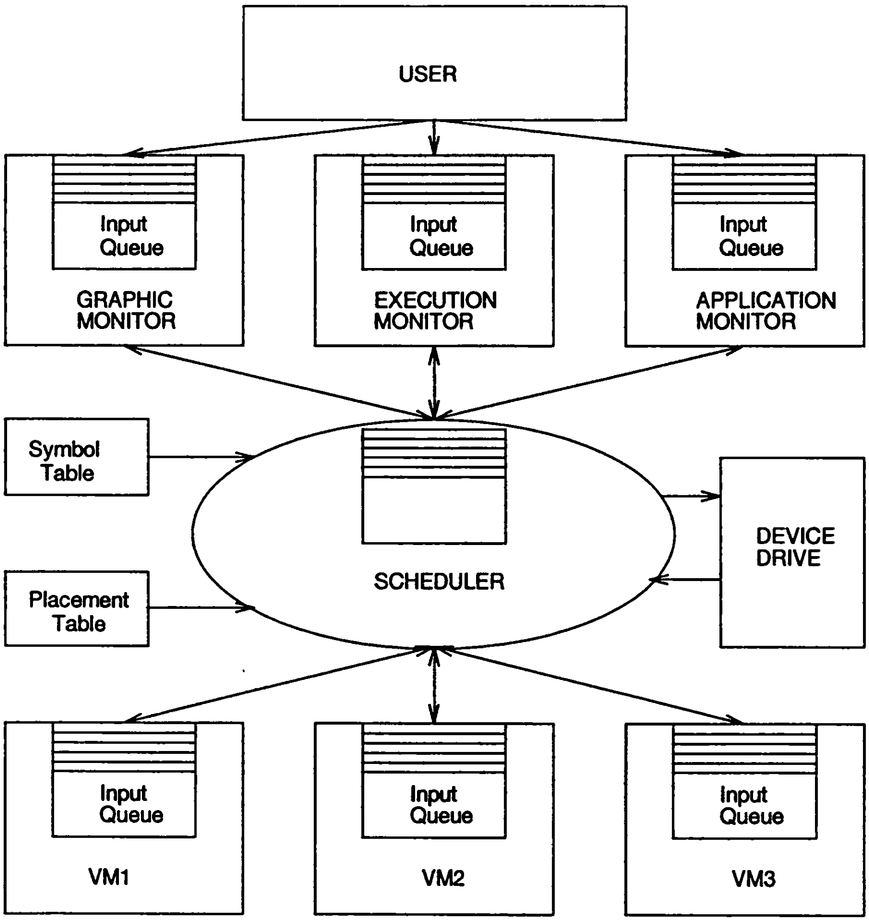


Figure 7.2. The Galatea Scheduler

The scheduler receives requests from users through the graphics, execution and debugging monitors which are independent processors as parts of the user interface. The requests go into a message queue and are prioritised before being processed. The Placement Table which was prepared by the mapper, is used to determine which *VML* concept (data) resides in which VM, and which *VML* rule is executed on which VM. The Symbol Table keeps *N* and *VML* concept-correspondence which are lost during the compilation from the high level language.

The GPNC is fundamentally a MIMD architecture with a message passing communications protocol. All modules of the GPNC and the scheduler use a generic message structure to send and receive messages, data and code. Messages contain source and destination fields, message type information as explained above, priority of the message, time of origination, size of the data and the data themselves. The generic message structure is as follows:

```
typedef struct message {
     int      source;
     int      destination
     int      type1;
     int      type2;
     int      priority;
     struct   timevaltime;
     int      size;
     char     *data;
}
```

The communication protocol consists of 19 different message types. The Table 7.1 explains these message types with respect to the data they contain. Two type fields identify each message. The first field, Field1, groups the messages into 5 main types of messages. These are responses, rules, data, control and graphics related messages. The second type field Field2 specifies the character of each message within these 5 groups. Response messages are issued as acknowledgements or progress reports etc. Rule messages addresses control issues at the rule level, for example to execute, continue or interrupt a rule. Data messages handle data transfer issues such as sending or requesting data. Control messages are used to set the debug level in execution or to stop (kill) execution of a certain VM or the whole application. It entails the level of 'kill' required. The last message type covers graphic commands, which are used in setting up the graphics canvasses and plotting graphs, bar charts or displaying data.

Field1	Field2	Description	Data
0		Responses	
	0	Acknowledgement	none
	1	End of rule	rule name, return code
	2	Exception condition	type, value
	3	Progress message	string
	4	Statement	statement type
1		Rule	
	0	Definition	code
	1	Execute	rule name, option
	2	Pause	rule name
	3	Continue	none
	4	Trace on	rule name
	5	Trace off	rule name
	6	Interrupt	rule name, option
2		Data	
	0	Definition	code
	1	Request	sync, block, freq, type, variable, rang
	2	Send	variable, range, value(s)
3		Control	
	0	Kill VM	
	1	Set debug	level
4		Graphics	
	0	Graphics: canvas	incanvas, title, x_axis, x1, x2, y_axis,
	1	Graphics: line	initcanvas, index, type1, type2 [, string]
	2	Graphics: plot	canvas, index, x, y [, string]

Table 7.1. Message Types

The message types and the message structure are used by all the VMs, GVMs and the Scheduler. Currently the data exchange and graphics plot statements in *VML* implicitly use the message protocol outlined above. The scheduler is based on a Message Passing Communications Scheme. All modules of the GPNC and the scheduler contain input request queues which are processed continuously depending on the message priority and type. All messages, which are put into the socket are placed into an input queue. The input queues are basically temporary memory storage for messages. A queue handler checks the message priority and executes the one with the highest priority. If there is more than one message with the same priority it fetches the oldest message. After a message has been found, copied and executed that message is removed from the input queue. This is especially important when carrying out user requests, but similar processes can be used to handle VM-Scheduler communications.

This section is presented to demonstrate the run-time operations on the GPNC simulator. The specification of the scheduler is the work of the author. The definition of the message types has been achieved in collaboration with a colleague at UCL, and the simulator of the Scheduler has been implemented later, by the same colleague. Graphical Virtual Machines have also been developed at UCL, by another colleague, to complete the GPNC simulator.

7.4. Mapping and Scheduling

A number of options are available for mapping and scheduling an execution on the distributed memory, message passing GPNC architecture. First of all, the following data-transfer protocols are considered between different modules of the system.

1 - Nonblocking write - The transmitting VM simply sends the data to the scheduler which in turn transmits the message to its destination. The transmitting VM continues the execution without confirmation from the receiving end.

2 - Blocking write - The same as above, but the transmitting VM continues execution after it receives confirmation that its message has been received.

3 - Nonblocking read - If the data request is not matched by immediate delivery, the VM continues with execution using its current copy of the same data. This mode can be used to simulate asynchronous models.

4 - Blocking read - When a data request is made by the VM the execution is paused until the required data is received.

The most practical combination is a nonblocking write matched by a blocking read. To implement this, *VML* is extended to incorporate the following data exchange commands on the TCP/IP server.

```
put_data ( destination, data_name )
get_data ( source, data_name )
```

The source and destination above relate to GVMs and VM, any data sent to a GVM is displayed by default. To plot any graph the graphical command is:

```
plot ( destination, x, y, index, canvas )
```

Here, the destination is a GVM, x and y are the axes, the index is a number between 0 and 9, referring to the dataset which will be displayed on a canvas. Up to 20 canvasses can be opened by a GVM.

Two radically different scheduling options are available. An active scheduler view supports the scheduler as the master process, controlling all execution, and data transfer instructions to the slave VMs. A passive scheduler view is more suitable for the distributed memory, multicomputer architecture of the GPNC. In this case, the Scheduler runs as a passive server process, unpacking messages originated by VMs and

GVMs, interpreting them and taking the necessary action.

7.5. Galatea Mapper Implementation

To implement the GPNC simulator on SUN workstations, the first step is to enable parallel simulations on a SUN local area networks (LAN). For this purpose Unix TCP/IP sockets have been used as the communications medium. TCP/IP is widely available on SUN and DEC workstation LANs, providing reliable, flow-controlled two-way transmission of data and messages. Initially, a prototype was developed to test the reliability of the communications medium. This prototype was used in chapter 4, in the simulation of parallel Hopfield nets. Later, this prototype was incorporated into the GPNC simulator, and has become the core of a comprehensive server, which undertakes scheduling tasks.

An evolutionary approach was followed in the implementation of the Mapper. It has involved, the manual mapping of parallel *VML* code onto a multi-VM environment, and its execution. Later, a semi-automatic mapper was developed, and an automatic mapper has been planned.

The first step in the implementation of the mapper is manual mapping or programming. After assessing the execution and the communications requirements for a given application and hardware characteristics, parallel *VML* code is written for all the parallel modules involved in the execution. The inter-VM data dependency and the load balance must be estimated, and data transfer instructions must be explicitly written by the programmer. The data movement commands are written in such a way that they display a handshake pattern between different modules. That is, a *get_data* statement in one VM is matched by a *put_data* in the other. If inter-VM data dependencies are not followed properly, the execution might come to a halt as a result of an unmatched data request statement. The time-sequence of the events must be planned in advance for a successful multi-processor parallel execution.

The second step in the mapper implementation is semi-automatic mapping. This involves the mapper parsing the sequential (raw) *VML* definition and generating parallel *VML* code, following the user directives stored in the placement table file. This file provides the total number of parallel VMs in the GPNC configuration, and the instructions for where to map each *VML* rule in the raw *VML* listing. The task mapping is straightforward and user-driven. It relies on the users' mapping instructions, based on their analysis and judgement of the application. The semi-automatic mapper is a *C* program which carries out the following tasks:

- Parses raw *VML* definition,
- Reads user mapping directives,
- Identifies and forms *VML* rule objects,
- Generates parallel *VML* listings.

The final step in the mapper implementation involves the full automation of the parallel code generation process. The Automatic Mapper generates parallel *VML* code for all VMs in the GPNC configuration, based on the minimisation of the computational costs. Automatic mapping is done by calculating parallel processing and communications costs on all possible combinations of rule mappings on a coarse number of VMs. The partitioning that results in the minimum computational cost is selected, the rules are grouped, and ASCII listings of the parallel *VML* code are generated with correct data transfer instructions. The Galatea project chose the *VML* rules to be the lowest level objects to be mapped. To achieve that, the Galatea Mapper breaks the raw *VML* code into self sufficient rule objects, all with a data definition part and a rule body. The automatic mapper consists of the following modules:

- *VML* Parser
- Variable and Rule Analysis
- Calculation of the Computational Costs
- Parallel *VML* code generation

Now, let us examine the modules of the automatic mapper, and the parameters involved in the automatic generation of parallel code for the Galatea GPNC.

***VML* Parser** - The Mapper uses the same parser routines as the *VML* interpreter [25]. This approach reduced the workload, as it ensures that the mapper automatically follows the modifications in the *VML* syntax. As a result of the parsing, the mapper generates its internal model of the *VML* rule and data structures upon which it carries out the variable and rule analysis.

***VML* Variable and Rule Analysis** - MIMD machines suffer from a data dependency problem. A classification of variable types is necessary to identify which variables have to be transmitted to other VMs, in a parallel execution. The Mapper carries out a variable analysis in which all variables are classified into 4 different variable types:

- Constant
- Local
- Read
- Write

The Constant type needs to be transmitted only once, at the beginning. This type of variables stay unchanged throughout the execution. The Local type refers to temporary variables. The Read type needs to be received from the Scheduler (or other VMs). Finally, the Write type implies that a variable has been modified within that VM, and should be transmitted to the Scheduler or other VMs. Similarly, a rule analysis is carried out to reveal the rule dependency, and to build the rule hierarchy for all the subrules and the caller rules for each rule.

Computational Costs - Performance of the parallel execution, which is defined as the computational cost of the execution depends on two different parameters: Communications Costs and Processing Costs. The communications costs directly depend on the amount of data exchanged between various processors (VMs). Using the variable and rule analysis for a given mapping, data sizes are calculated for all the Read and Write type variables. The *VML* LOOP statement is decoded to obtain an approximate measure for the number of repetitions occurring for each command line. This measure is necessary to determine the volume of data which is transferred between the VMs and to establish the processing costs within loops. Real processing costs are hardware dependent. For a given architecture, the processing costs depend on the following parameters:

- Operation Type
- Data Type
- Data Size
- Placement Type
- Hardware Characteristics (speed and memory)

After identifying these parameters, estimates of the processing costs for each operation type were requested from Siemens and Philips hardware groups, with the intention to use the parameters as inputs for automatic mapping strategies. As a result, the following issues are highlighted;

1 - The optimal data type must be decided in *VML* code before or during the mapping. This is a very important issue for the hardware, and it is hardware-specific. For example

the Philips accelerator board achieves the highest performance on fixed point arithmetic operations.

2 - The optimal placement type must be defined for different data types in *VML*. The Siemens group [17] stressed the importance of the correct placement of data on the local memories of their VM, at the initial mapping stage. Four types of memory were reported on the Siemens hardware, namely: *wmem, ymem, zmem* and *cmem.* Siemens also listed the best memory placement for a number of data types as follows:

Placement Type	Memory Type
PLACEMENT_COMMS	
PLACEMENT_FREE	wmem
PLACEMENT_STATE	ymem
PLACEMENT_WEIGHT	wmem
PLACEMENT_PATTERN	ymem
PLACEMENT_LUT	cmem
PLACEMENT_INPUT	ymem
PLACEMENT_OUTPUT	ymem
PLACEMENT_TEMPORARY	ymem

3 - The calculation times for matrices of various sizes can be made available after the manufacturing and tests. Only approximate information was given about the clock cycles for certain matrix operations which will be executed on the VM hardware.

Following these developments the new version of *VML* (version 2.0), incorporated PLACEMENT_TYPE field in the matrix declaration. As *VML* has no concept of pattern or weights this field can be ideally decided by the *N* level and passed down to *VML* through the compiler. Mimetics notified that they could generate PLACEMENT_STATE, PLACEMENT_WEIGHT, PLACEMENT_TEMPORARY placement types using the *N* to *VML* compiler. Mimetics also suggested that INPUT, OUTPUT placement types can be determined in *N*, and hinted that new placement types could be necessary for optimal mapping. In addition to this, *VML 2.0* also includes the data type information prefixed to all *VML* instructions. In this way, *VML* becomes similar to low level VML (*LLVML*) [42] and the generation of the low level commands on the VMs, is made easier.

Mimetics automatically generated *VML 2.0* code for a number of neural network models including the Gradient Descent Backpropagation model. *N* to *VML* compiler is used in the automatic optimisation and generation of the raw *VML* code, completing the full cycle of GPNC automatic code generation. The cycle involves the *N* to *VML* compiler and the Mapper running together and generating parallel *VML* code for the VMs. The compiler generates the raw V*ML* code, the symbol table, and the configuration table. The Mapper generates parallel *VML* and the Placement table to indicate the rule placement on the VMs. This data can also be used if a re-configuration is required like in the case of Dynamic Mapping.

7.6. Mapping *VML* to Galatea GPNC

This section presents the Galatea Mapper results and simulations on the GPNC. The results of the manual, semi-automatic and automatic mappings are presented.

7.6.1. Co-operating Hopfield/Backpropagation Networks

Two simulations are designed to demonstrate the co-operation of two neural network models on configurations consisting of two or three parallel VMs. These simulations are examples of manual mapping or parallel programming on the GPNC simulator, with the aim of demonstrating the feasibility and the strength of the parallel techniques.

The simulations involve using a Hopfield and a Backpropagation networks in co-operation. The Hopfield net is used as an auto-associative memory unit reconstructing noisy or partially corrupted patterns. The Backpropagation network is trained to imitate the associations made by the Hopfield net. In this two-network architecture, the Hopfield net trains the Backpropagation network. The following advantages can be gained using this multi-network configuration. Firstly, the Backpropagation network is fast in recall and extremely slow during the training. The Hopfield net, on the other hand, which operates with fixed weights, is relatively slow during the recall, as the convergence takes place at this stage. Secondly, the Backpropagation model is more robust than the Hopfield net in recall. If the Backpropagation network can be trained to respond like the Hopfield net, later, the Hopfield net can be bypassed, and the Backpropagation network can be used in auto-associative recall tasks much more efficiently. Once the Backpropagation network is trained on a training set, it can be used more reliably in pattern classification tasks.

The multi-network configuration can be efficiently executed on two VMs with each network running in parallel on separate VMs. Both networks receive noisy inputs from the external world, and the Hopfield net provides the target patterns for the Backpropagation network. In this case the data exchange between the VMs is unidirectional. As an application, the same pattern recognition problem is outlined as the one used in the analysis of the Hopfield net in the chapter 4. The Hopfield net has fixed weights prior to the execution, and the Backpropagation network starts with small random weights.

From the scheduling point of view, this is an intertwined application requiring explicit data transfer instructions. This can be achieved by writing the application in *VML*, or semi-automatically partitioning the raw *VML* code. It is hard to achieve the same configuration through automatic partitioning and mapping. The difficulty lies in the automatic generation of data transfer instructions as they are intrinsic to the algorithms. Users either write the *VML* code with relevant data transfer instructions, or alternatively, they can supervise parallel *VML* generation by putting mapping instructions in the placement_data file.

The simulation shows data exchange and cooperation between a number of VMs in solving a problem together. Two Graphical Virtual Machines accompany the two VMs, to coordinate the graphical display operations required by the VMs. Together with the Server, 3 VMs and 3 GVMs, a total of 7 processes run concurrently as independent Unix processes, communicating with each other through TCP/IP sockets. The server opens a socket communications channel, and all other processes plug into this medium. In the run-time, the server decodes messages sent by VMs and GVMs, routes them to their appropriate destinations. Two different scheduling methods have been tested on the Hopfield/Backpropagation cooperating networks simulations involving an active and a passive scheduler.

Demonstrator I

VM1 acts as an active Scheduler organising data movements between the other VMs (Figure 7.3). The *VML* code organises the handshake for data transfer operations in the following way. All *put_data* statements are nonblocking and the matching *get_data* statements are blocking.

```
         VM2                VM1/Scheduler                VM3
put_data (1, matrix1) -> get_data (2, matrix1)
                         put_data (3, matrix1) ->  get_data (1, matrix1)
```

The duties of the three VMs in the Hopfield/Backpropagation co-operating neural networks example are:

1 - VM1 is the Scheduler. It has access to the system file storage devices. All the file I/O operations are performed using this VM. It coordinates data exchange between the other two VMs. The execution in the three VMs is started simultaneously. VM1 loads the trained weights for the Hopfield net from the file store, sends them to VM2. And VM3 initialises its weights with small random values. VM2 waits for an input pattern to recall and reconstruct, and VM3 waits for the training pair to arrive. VM1 reads the incomplete or noisy input patterns and routes them to VM2 and VM3. After this, it receives reconstructed patterns from VM2 and reroutes them to VM3 which starts the Backpropagation network training. After a training run, it receives the resulting Backpropagation network weight sets from the VM3, and saves them onto the file store.

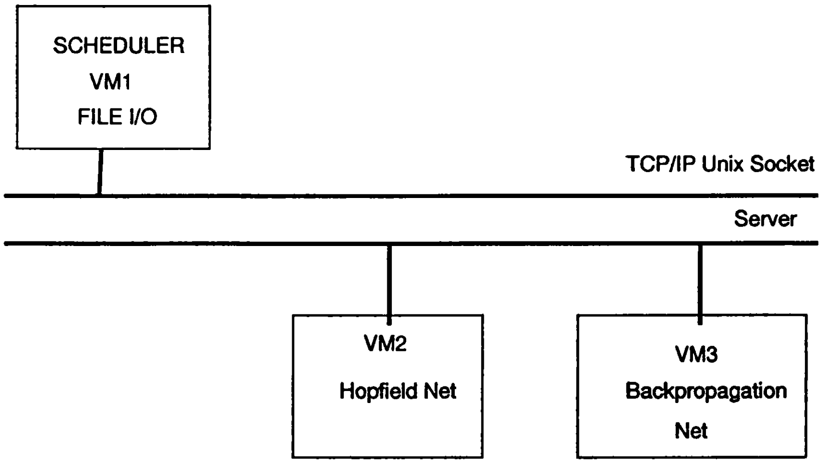


Figure 7.3. Demonstrator I

2 - VM2 is a Hopfield network. It receives the trained weights and the incomplete input patterns and starts a Recall operation. As a result it reconstructs the patterns and sends them to the VM2.

3 - VM3 is a Backpropagation network. It receives the same inputs as the Hopfield network, and then as it receives targets, reconstructed by the Hopfield network, it trains with the pairs it received, as it receives. When it receives all the targets it trains for all the patterns, in the end transmitting the results back to the Scheduler.

Demonstrator II

In this version, the Scheduler is a passive process that allows VMs to communicate with each other directly by addressing one another (Figure 7.4). This way of scheduling creates intertwined *VML* listings. There is no need for a separate program as the scheduler passively enables the inter-VM data communications. Messages are received by the Scheduler and the data are routed to their correct destinations on the socket communications. It is hard to generate this kind of parallel mapping automatically. Manual mapping enables us to design such an application with the correct handshake between the *get_data* and *put_data* statements, and it is hard to see how such an application could be generated automatically. The *VML* code in the two VMs is as follows:

```
               VM1                           VM3

      put_data (3, matrix1) -->   get_data (1, matrix1)
```

This time VM1 is not the Scheduler, but it is like the other VMs; an ordinary VM only with an access to the file storage. To start the demonstration, first the server is run, then all the VMs connect to it. Again, 7 processes run on the Unix network. All requests generated by the VMs are put into the Scheduler input queue which are processed according to their priorities defined in the message structure.

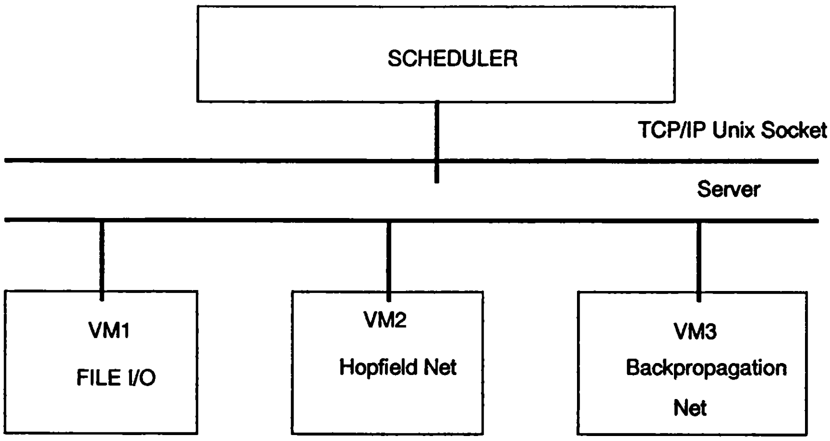


Figure 7.4. Demonstrator II

Each VM is accompanied by a Graphical Virtual Machine providing interactions with the Scheduler and the external world. In interactions with the external world (Unix environment) X Windows Event Handler is used to detect user requests from the graphics and windows environments.

7.6.2. Semi-Automatic Mapping of Backpropagation

This simulation involves mapping the Backpropagation Recall rule onto a high performance *VML*. The rest of the execution is contained on the host. The manually generated Backpropagation VML listing consists 4 rule bodies:

Rule 0:	Randomise
Rule 1:	Recall
Rule 2:	Learn
Rule 3:	Main

On a 2 VMs configuration the Placement table instructs the mapper to place the Recall rule and data definition to the VM2. This table, in a file contains the following data;

```
2         #Number of VMs
0    1    #Rule 0 to VM1
1    2    #Rule 1 to VM2
2    1    #Rule 2 to VM1
3    1    #Rule 3 to VM1
```

The semi-automatically mapped parallel Backpropagation simulation on two SUN4s ran 90 times slower than running sequentially on a single SUN4. That is a 1 minute-long sequential simulation took approximately 90 minutes. These results not encouraging, and they are due to the low communications throughput provided by the scheduler on the SUN LAN.

7.6.3. Automatic Mapping

Galatea Project viewed the automatic mapping of secondary importance, as an optimisation issue, and it focused on the efficient execution of VML on the generic boards of Siemens and Philips VMs. Only approximate hardware characteristics could be used at the time, as VM boards have not been built. The experiments with manually mapped Cooperating Networks and the semi-automatic mapping of the Backpropagation model examples show inter-VM communications present the highest share of the total computational cost. Considering this is only a software simulation, and the Galatea GPNC will have a much faster communications medium, the resulting system might be much more rewarding. When the real performance figures on the hardware are available, the mapper can use these parameters for an accurate mapping.

7.7. Summary

This chapter presented the Galatea Mapper and Scheduler as part of the Galatea GPNC simulator. The Galatea Mapper has been designed, developed and tested as part of the Galatea automatic code generation process. The Galatea Mapper focused on the manual and semi-automatic generation of parallel *VML* from a sequential definition.

The manual mapping of the Hopfield/Backpropagation model, and and the parallel simulations developed as part of this research, were displayed as the demonstration for the Galatea GPNC, in Brussels during the Esprit week, in October 1991.

Semi-automatic mapping of the the Backpropagation model onto two SUN workstations, and the consequent parallel simulation resulted in a 90 times slower execution than the sequential simulation of the same application. This was due to the overheads on the Scheduler, and the slow communications medium provided by the Ethernet.

The semi-automatic Galatea Mapper does not meet the requirements of high performance, flexibility and generality. It requires optimum rule definitions and explicit mapping instructions from the user. Automatic mapping was left out of the Galatea GPNC development as an optimisation issue. As one of the objectives of this thesis is to achieve an automatic mapping strategy, it is further investigated in the mapping of *MATLIB* representations in chapter 8.

Chapter 8

Mapping MATLIB Representations

This chapter presents the computational analysis tool projections and parallel MATLIB simulations on a SUN network. First, the CAT results are verified, then the three neural network models are parallelised, finally, parallel multiple neural network simulations are presented.

8.1. Simulations Overview

The main objective of parallel mappings is to enhance neural network performance and provide a scalable, efficient execution on parallel hardware platforms. To achieve this objective, a general-purpose execution strategy has been adopted, which exploits general-purpose parallel neurocomputing modules called Virtual Machines. A matrix-based *C* library, *MATLIB* has been designed and developed as a suitable representation for parallel mapping onto VMs. The mapping strategy has been outlined as the optimisation of computational costs on a number of parallel processors. To apply the mapping strategy on *MATLIB* representations the CAT and APM have been designed and developed. The efficiency of mappings depends exclusively on the proper computational analysis of neural network representations and assessment of parallel hardware characteristics.

It is then necessary to verify the CAT results, as the mapping decisions depend on its performance. The first part of this chapter is dedicated to the validation of CAT simulation results. For this purpose, CAT was used to analyse the *MATLIB* listings of the three models, the Hopfield, the SOM and the Backpropagation. A number of neural network and hardware configurations were used in the simulations on a number of SUN workstations assumed as a number of parallel VMs. CAT results were then compared with results from standard Unix timing facilities.

The Computational Analysis Tool has been implemented as an integral part of the parallel mapping process with an intention to automate the mapping. However, the primary use of CAT is in the identification of computational bottlenecks and potential data and task parallel executions.

For data parallelism, the computational cost of a potential data parallel execution is estimated for each instruction. The estimate is then compared with the potential sequential cost on the host. Based on this comparison, a mapping decision is made and finally, the overall parallel cost is estimated.

Task parallelism, or a potential pipelined execution on parallel hardware is also explored by CAT through a variable/loop analysis. The communications costs are estimated in the case of splitting *MATLIB* representations for an instruction-pipeline type parallel execution.

The CAT results are used both as a guideline for parallel programming and in the automation of the mapping process. Using CAT, the three models were partitioned, and simulated on a number of SUN workstations Local Area Network. Parallel simulation results were timed and compared with sequential execution results.

Finally, as the main objective is to enhance neural network performance on parallel hardware, *MATLIB* and *NETLIB* parallel libraries were used in mapping and simulating multiple neural network models on parallel platforms. A number of complete neural networks were mapped onto separate parallel processors as cooperating and competing modules. These simulations are also examples of efficient algorithms that can exploit coarse-grained general-purpose parallel computers.

8.2. Computational Analysis Tool Results

This section presents the results of the CAT simulations on *MATLIB* representations of the three neural network models. The main objective in these simulations is to validate the CAT computational modelling approach on a SUN LAN, and to demonstrate that the technique can be used reliably in automating the parallel mapping process. In addition to this, a secondary objective is to assess computational characteristics of the three models. To achieve an approximate computational model of the hardware, the processing and the communications costs must be parameterised. A number of simulations was used to parameterise the processing and communications costs of the SUN LAN.

Firstly, a computational look-up table is set up, which consists of the computational unit costs of all *MATLIB* functions. Each *MATLIB* function is executed using various data size and repetitions on a SUN4 workstation to establish the unit cost for each operation. Table 8.1 shows the estimated unit costs for each matrix operation. These costs are multiplied by the data size to calculate the operation cost for an instruction, and the operation cost is multiplied by the repetition of that line to estimate the cost for that code line. The repetition is a result of the 'for' loops encircling that specific line, and is obtained by the loop analysis. By adding all the line costs, the overall execution cost for the potential execution of the *MATLIB* listing is estimated. A linear model is assumed for simplicity, so the computational cost is a linear function of the data size. The data size is always the actual processed data size, rather than the destination matrix size. For example, in the case of *mmul*, the multiplication of $A(2,10)$ by $B(10,1)$ matrices results in a column matrix of $R(2, 1)$. To use the destination matrix size, *2* as the data size would be a massive underestimation of the computations involved. Instead, the data size is calculated for each type of operation separately. For the above example, the data size would be the multiplication of the output matrix size by the column size of the first A matrix, which is *10*. Similar considerations are made for other operations. In the case of *mmin* and *mmax* the computational time varies depending on the execution of the conditional 'if' statements in these operations. For this reason, the computational look-up table is not 100% accurate, but it provides a good approximation for computational evaluation purposes.

MATLIB Functions	Description	Unit Cost *μsec*
mmul	matrix multiplication	8.0
madd, vadd	matrix/vector addition	6.0
msub, vsub	matrix/vector subtraction	6.0
memu, vemu	matrix element multiplication	6.2
mscm	matrix scalar multiplication	6.2
mtan	apply tanh to matrix	36.0
dtan	apply derivative tanh to matrix	12.0
msig	apply sigmoid to matrix	21.4
dsig	apply derivative sigmoid to matrix	6.5
mrms	matrix root mean square	8.0
mmax, mmin	matrix maximum/minimum	5.0
mabs	matrix absolute	5.0
mavg	matrix average	6.0
mlat	lateral weight update for SOM	9.0
mcpy, vcpy, ccpy, mtra	matrix copy, transpose	4.5
mset, sval, msal	matrix element set/get	4.5
mran	matrix randomise	7.7

Table 8.1. Computational costs for *MATLIB* on sun4

Secondly, unit communications costs are established for the network of SUN workstations. A number of simulations have been carried out to model the average communications cost. The results in Figure 8.1 show a quasi-linear model for the communications costs. As the data size increases, transmission time increases. Various sizes of matrices have been transmitted through the sockets. The transmission times have been measured using *MATLIB* communications statements, *put_mat* and *get_mat.* An overhead of 0.5 seconds is found for opening up a socket and connecting to an open socket. In addition to this, as the sockets are packetised communications media, there is a minimum cost of 0.1 seconds regardless of the transmitted matrix size. Considering that the purpose of the parallel simulations is to assess the mapping strategy, rather than achieving a high performance on the Ethernet, these extra costs are negligible, and a linear model is sufficient in this case. The communication speed for the *MATLIB* data transmission on the SUN network was found to be 180,000 bits/second. In other words the cost of transmission is 1/180,000 sec/bit. This approximate figure includes time delays during the packing of messages before transmission, and the unpacking that takes place after the reception.

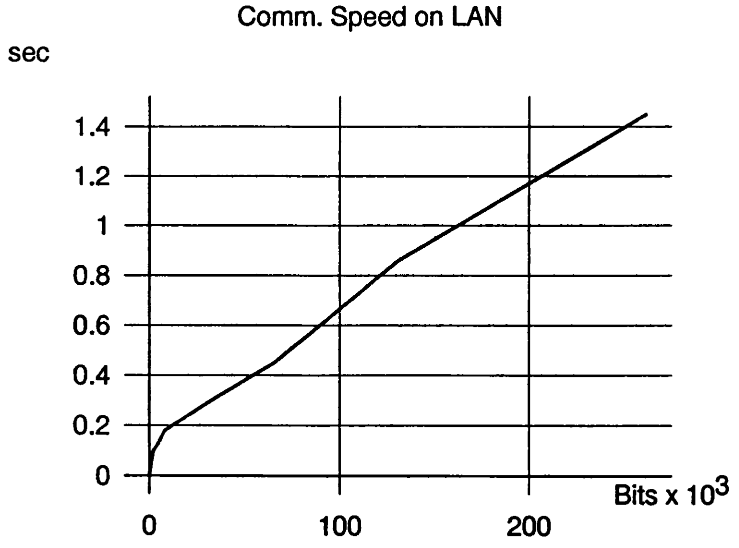


Figure 8.1. Socket Communications Costs

A number of simulations are carried out to verify the results of the Computational Analysis Tool simulations on the Hopfield, the SOM and the Backpropagation *MATLIB* representations with different topologies and parameters. These simulations also show the computational sensitivity of the models to the parameters of the networks. CAT results shown in Figures (8.2, 8.3, 8.4) are very close to the actual results obtained by timing the executions. In fact, the method of timing the executions using Unix commands is not as reliable as CAT results, as the timing method depends on the processing load of the workstation, the ethernet traffic, and the fileserver load, at the time of the execution. The Computational Analysis Tool, on the other hand, produces estimates of execution times regardless of these parameters, based purely on the computational unit costs and data size.

The first set of results for the Hopfield model show a polynomial increase in the computational cost (and the requirements), as the number of neurons increases linearly (Figure 8.2). This is a result of the *mmul* operation whose computational cost also shows a polynomial increase as the number of neurons increase. As the number of neurons N increases, computational time t increases rapidly, following the polynomial relationship $t = N^2$. This is in line with expectations, as the Hopfield weight matrix size follows the same relationship when the number of neurons are increased.

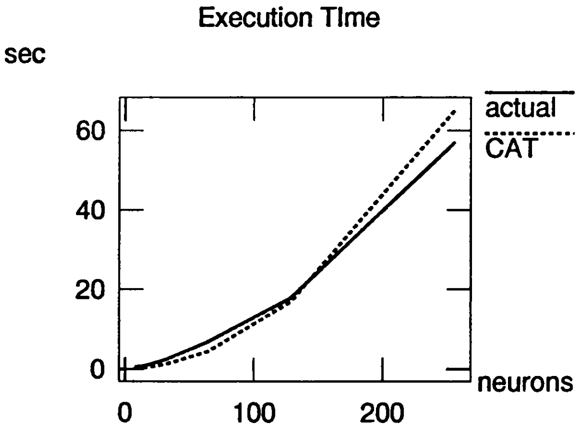


Figure 8.2. Actual and CAT results in Hopfield Net

The results for the SOM model show a linear increase in the execution time, when the number of neurons in the input layer is increased (Figure 8.3.a). The SOM is most sensitive to the increase in the number of neurons in the output layer (Figure 8.3.b). In this case the computational requirements increase polynomially as a result of the enlarged output grid. Again, as expected, computational requirements polynomially increase with the increasing number of output nodes.

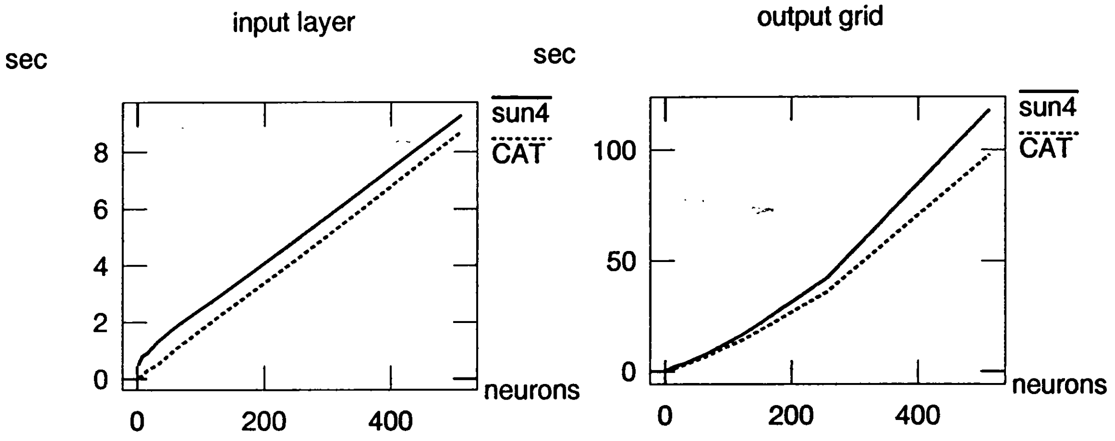


Figure 8.3. Actual and CAT results for the SOM

The results for the Backpropagation model show a linear increase in the execution time, when an increase is made in the number of training patterns or the number of neurons in the input layer, hidden layer or output layer (Figure 8.4).

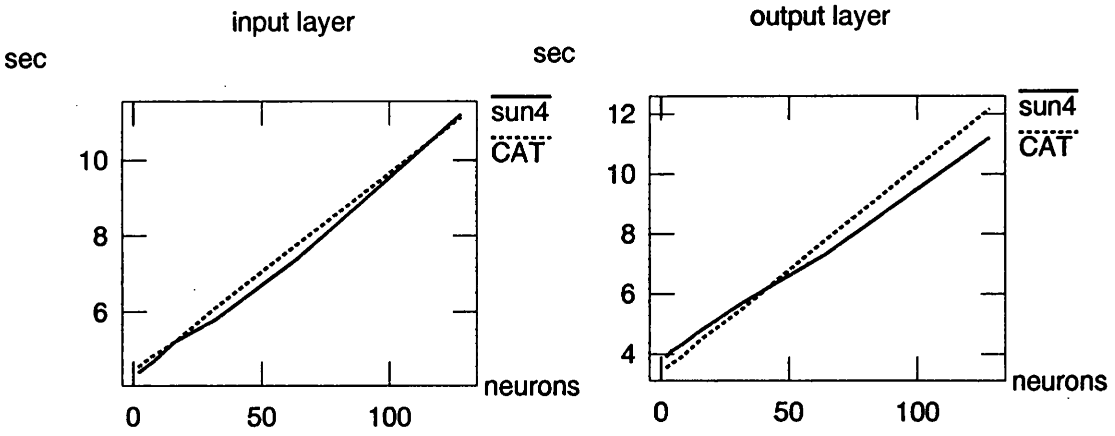


Figure 8.4. Actual and CAT results for the Backpropagation

An accurate computational analysis is important for the mapping decision which is based on the computational optimisation of the use of parallel resources. CAT provides very close results to the actual results obtained by standard Unix *time* function, and CAT results can be reliably used for the computational analysis of any algorithm coded in *MATLIB*. Other CAT functions, such as variable and loop analyses, parallel and pipelined communications cost calculations are shown in the parallel mappings of the three neural network models.

8.3. Mapping the Three Models

In this section, the three neural network models which have been the centre of focus throughout this thesis work, are used in data parallel and task parallel mappings and simulations. The *MATLIB* representations of these models are processed by CAT, and simulated onto a number of parallel SUN4 workstations. The same pattern recognition problem is used in all these simulations to assess the performance of the three models and their relative computational requirements for the same problem.

In addition to dumping total memory use and sequential computational costs on the host, the CAT results point to computational bottlenecks and possible parallelism in sequential *MATLIB* programs.

8.3.1. Hopfield Nets

This set of simulations aims to parallelise the Hopfield net and reduce execution time through parallel execution. The same dataset and the 64-neuron topology with the Hopfield net case study in chapter 4, are also used here. The net is initialised with fixed weights and is used in recalling the original patterns from the noisy or incomplete inputs. A line by line computational analysis of the Hopfield net results are shown in Table 8.2.

line	function	repetition	data size	unit cost *µsec*	op. cost *µsec*	line cost sec	total cost sec
24	*vcpy*	12	64	4.5	289	0.003	0.003
26	*mmul*	48	4096	8.0	32768	1.573	1.576
27	*mtan*	48	64	36.0	2307	0.111	1.687
28	*msub*	48	64	6.0	384	0.018	1.705
29	*mrms*	48	64	8.0	512	0.025	1.730
33	*mcpy*	48	64	4.5	289	0.014	1.744
37	*vcpy*	12	64	4.5	289	0.003	1.743

Table 8.2. CAT results for the Hopfield net

Using these results, a computational bar chart is drawn in Figure 8.5, which shows the line by line computational cost on the Hopfield *MATLIB* listing. According to these results, most of the computational time is spent in the matrix multiplication *mmul* operation, amounting to almost 90% of all computations for a 64-neuron, 12-pattern Hopfield net simulation on a SUN4 workstation. This is not surprising considering the data size *mmul* has to process (Table 8.2).

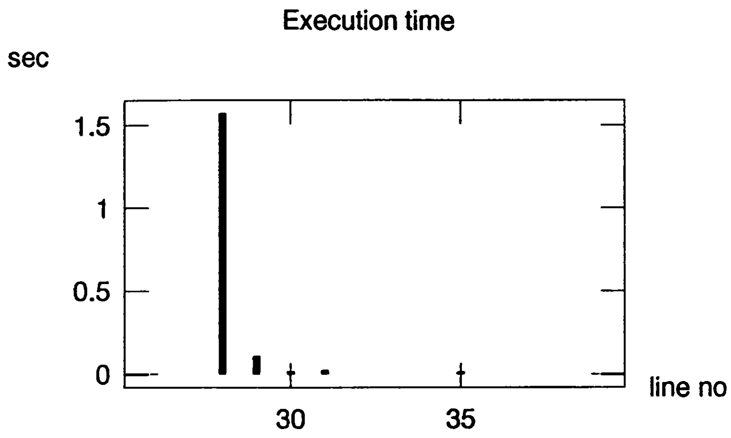


Figure 8.5. Line by Line Computations in Hopfield Net

Task Parallelism in Hopfield Net

CAT variable/loop analyses reveal that, in the main loop, where the network convergence takes place, there is a backward data flow on matrix variable Temp. Table 8.3 shows the line by line variable analysis for Temp. In this table, crosses indicate read and write operations on this matrix, and the resulting forward and backward data paths. The backward data stream prevents the instruction pipeline type of parallelism, as the beginning of the loop cannot progress without reading data written at the end of the same loop, for example in the case of dividing the representation into two. Another reason for not favouring task parallelism is that the load balance is impossible on a prospect pipeline, as the matrix multiplication takes most of the computations for the Hopfield net. This leaves data parallelism as the only viable alternative.

line	instruction	Backward Flow	Forward Flow	Read	Write
25	*for*		✗		✗
25	*for*		✗		
26	*mmul*	✗	✗	✗	
27	*mtan*	✗	✗		
28	*msub*	✗	✗	✗	
29	*mrms*	✗			
30	*if*	✗			
31	*break*	✗			
32	*endif*	✗			
33	*mcpy*				✗
34	*endfor*				

Table 8.3. Variable Analysis for Temp

Data Parallelism in Hopfield Net

CAT can estimate both sequential and data parallel computational costs on a line by line basis, and these estimates can be used in a parallel mapping decision. The calculation of the parallel processing and communications costs is based on the parameterised data parallel cost model presented in chapter 6. For each *MATLIB* line or instruction, the estimated parallel computational cost is compared with the sequential cost on the host, which is again calculated by CAT. If the potential parallel execution cost is less than the sequential cost on the host, the operation is parallelised. Otherwise a sequential execution on the host is preferred. Table 8.3 shows sequential versus parallel computational costs for all *MATLIB* instructions on the 64 neuron-Hopfield net, with 12 patterns, on 2 parallel SUN4 stations linked with a 10 Mbit/sec speed bus. The mapping decision which is made on a line by line basis, depends on the unit cost, data size, the number of parallel modules and the communications speed. For example, in line 24, the *vcpy* vector copy operation is less costly on the host, so this operation is not partitioned. The most expensive operation *mmul* in line 26 will be executed quicker in parallel, so the mapping decision is made favourably. Considering that each operation is repeated many times, the overall gains are greater than the gains made in single operations. For the example shown in Table 8.4, CAT calculated the sequential execution cost as 1.74 seconds, and projected the parallel execution as 1.56 seconds, for the same Hopfield configuration mentioned above on 2 parallel processors connected to a host by a 10 Mbit/sec bus. The computational characteristics of the parallel processors are assumed to be similar to the host, a SUN4 workstation. This performance is assumed to be gained by parallel mapping of the three *MATLIB* functions shown in the table.

line	function	seq op cost *μsec*	par op cost *μsec*	mapping decision
24	*vcpy*	289	2705	-
26	*mmul*	32768	29798	✓
27	*mtan*	2307	1460	✓
28	*msub*	384	704	-
29	*mrms*	512	461	✓
33	*mcpy*	289	452	-
37	*vcpy*	289	2705	-

Table 8.4. Data Partitioning on Hopfield

Using the projected parallel execution costs, comparisons can be made with the sequential cost on a number of bus architectures with varying communications speeds and a number of parallel processors. Figure 8.6 shows parallel execution costs for the same Hopfield net on varying communication speeds and the number of VMs. As all mapping decisions are negative for slow speed buses, the maximum execution time does not exceed the sequential execution time. In these cases, operations are not partitioned, instead, they are executed on the host. Assuming the SUN4 computational characteristics, parallel mappings of the Hopfield net onto architectures with communications speeds below 4 Mbit/sec result in no speed-up in parallel executions. These results show that, on parallel architectures, the communication speed is one of the most important parameters.

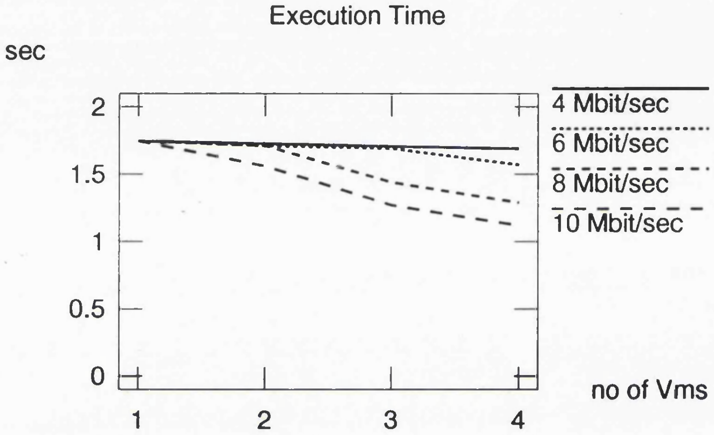


Figure 8.6. Projections on Data Parallel Hopfield

Further parallel projections are carried out involving Hopfield nets with varying number of neurons and VMs. These projections assume a 10 Mbit/sec communications speed, and the nets are iterated a fixed number of cycles with no convergence requirements. Results in Figure 8.7 indicate that, as the number of parallel processors increases, the parallel executions get progressively shorter. Although the performance is improved by increasing the number of parallel processors, the efficiency is another matter, which is discussed in chapter 9.

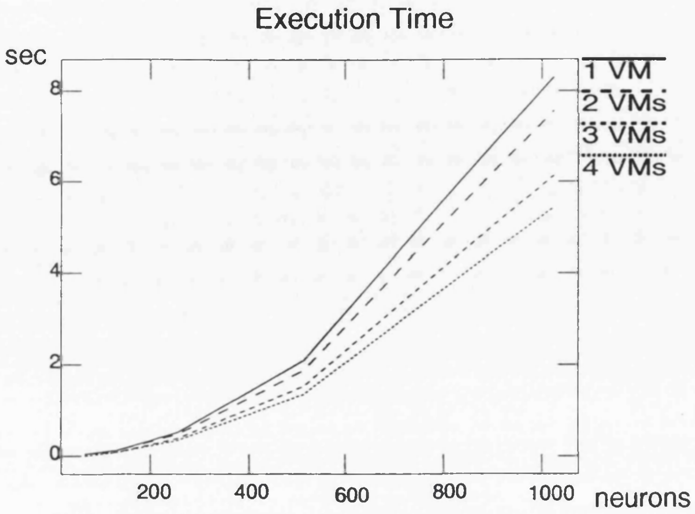


Figure 8.7. Scalability of Data Parallel Hopfield Executions

Data Parallel Simulation of Hopfield on the LAN

Based on the computational analysis on the Hopfield net (Appendix D.1), it is shown that the net is computationally most sensitive to an increase in the number of neurons. If networks with a large number of neurons are simulated on sequential machines, computational bottlenecks are unavoidable. The backward data path mentioned earlier, does not allow instruction pipelining, leaving data parallelism as the only effective parallel mapping option for the Hopfield net. In fact only parallelising the matrix multiplication would be a big gain for the execution. Parallel *MATLIB* features can be used to partition the matrix multiplication operation, and the partitioned operation can be executed in parallel on a number of SUNs. A number of simulations have been carried out to test this approach.

First, the *mmul* operation has been parallelised on two parallel SUN4s. To do this, parallel code for the main Hopfield rule body, and a number of identical matrix multiplier *MATLIB* modules are written with explicit data transfer statements (Figure 8.8). Matching and blocking data transfer instructions are used to transfer data between the main body (the scheduler) and a number of parallel multipliers (clients). The scheduler program divides the weight matrix into *n* equal parts and transmits the submatrices to the clients (in this case 2 parallel clients are used). The equal parts of the weight matrix HW needs to be sent to the clients only once, at the beginning of the execution. As it is constant, it can be stored in local memories of the clients. Then, the scheduler sends parts of the row matrix ST0 to the clients, which wait for data to arrive. ST0 is an argument of the matrix multiplication, and must be transmitted prior to the operation. As soon as the clients receive partial ST0 data, they all carry out matrix multiplications, and on completion, transmit the resultant row matrix. The scheduler reassembles submatrix multiplication results by adding the partial results to form the overall result matrix. This addition operation is an overhead, an extra computational load on the scheduling processor, which does not occur during sequential execution. This is due to the high-level nature of the matrix multiplication operation which involves a series of multiplications followed by additions. The overhead introduced by *n* parallel multipliers is *n-1* matrix additions.

```
           The Scheduler                                  The Clients
           -------------                                  -----------
/* Transmit half the HW */                   /* Receive half the HW        */
put_rows( vm[1], HW, 0,      SIZE/2 );              get_mat( HW, fd );
put_rows( vm[2], HW, SIZE/2, SIZE );

/* Transmit half the ST */                   /* Receive half the ST        */
put_cols( vm[1], ST0, 0,      SIZE/2 );              get_mat( PP1, fd );
put_cols( vm[2], ST0, SIZE/2, SIZE );

                                             /* Carry out Multiplications */
                                                    mmul( ST1, PP1, HW );

/* Receive partial results */                /* Transmit result            */
get_mat( ST1, vm[1] );                              put_mat( fd, ST1 );
get_mat( ST2, vm[2] );

/* Sum partial results */
madd( ST, ST1, ST2 );
```

Figure 8.8. Scheduler-Client Interaction

As CAT reveals, the parallel simulations on a network exploit the fact that the weight matrix is a constant, and needs not be transmitted for every matrix multiplication. Currently, CAT's line by line parallel projections cannot exploit this, as parallel mapping decisions are localised for each instruction and a temporal, global variable analysis scheme is necessary. In fact, the distribution and the local storage of the weight matrix, makes parallel executions on the SUN LAN faster than the sequential execution. The results in Figure 8.9, show that parallel simulations of the Hopfield nets with various number of neurons, is faster on two parallel SUN4s than the sequential execution on a single SUN4.

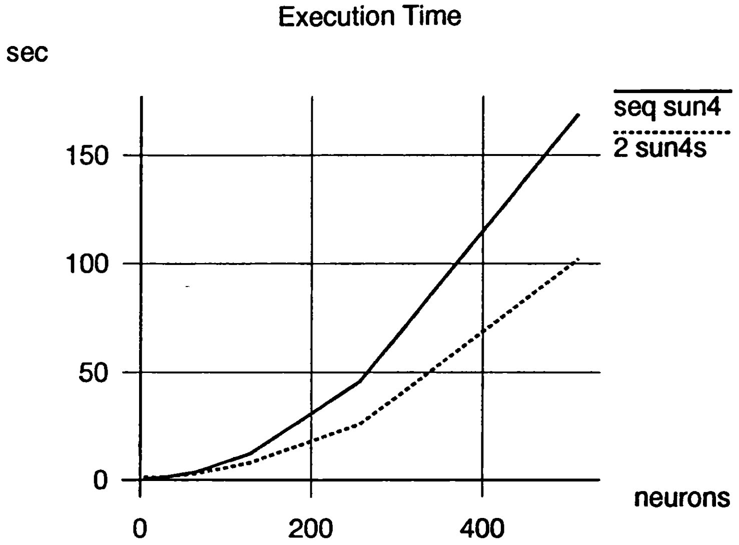


Figure 8.9. Sequential and Data Parallel Hopfield

These results are remarkable considering that the communications speed for the parallel simulations environment is as low as 180,000 bits/sec due to the high-level data transfer routines. In all these Hopfield simulations, varying number of neurons with randomly generated weight and pattern matrices are used with no convergence requirements, focusing only on the execution time. Following these simulations on two SUN4s, a 300 neurons Hopfield net is simulated on 2, 3 and 4 SUN4 workstations with the same data parallel method. These simulations show the execution time is reduced considerably for 2, 3 and 4 processors (Figure 8.10), and it tends to increase after 4 processors. This is because of the overheads in the opening up a socket, connecting to the socket, and the standard costs relating to communications. These increased communications costs prevents a linear curve of reduction in the execution time, when the number of VMs increase. These results also highlight the importance of initial communications costs such as latency which is constant and independent of the data size.

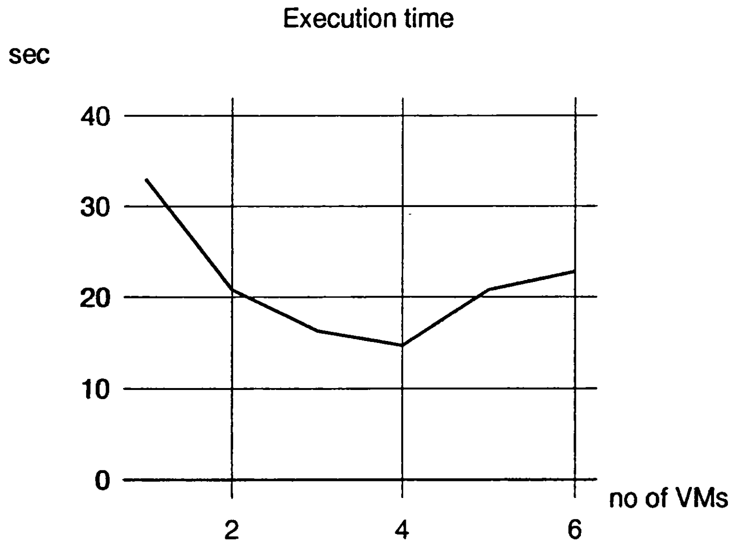


Figure 8.10. Hopfield on multiple processors

These results confirm that data parallelism is feasible for the Hopfield net, with potential gains in performance on parallel hardware. The results also indicate that the execution time can be reduced only by dividing the computationally intensive matrix multiplication operation into a number of parallel submatrix multiplications. CAT can detect this in *MATLIB* listings, and a data parallel execution is possible using *MATLIB* representations with no modifications. Based on these results, the mapping operation can be automated by adding a data parallel code generation module to APM, and this module can be triggered by the computational analysis and the subsequent mapping decision which is made by CAT.

8.3.2. The Self-Organising Maps

Similar to the Hopfield net, the possibility of data and task parallelisms for the SOM has been explored. CAT provided a line by line computational profile (Figure 8.11). In this case, the distribution of the computations is more even, with some operations such as matrix root mean squared (*mrms*) and lateral matrix update (*mlat*) relatively more demanding than the other calculations.

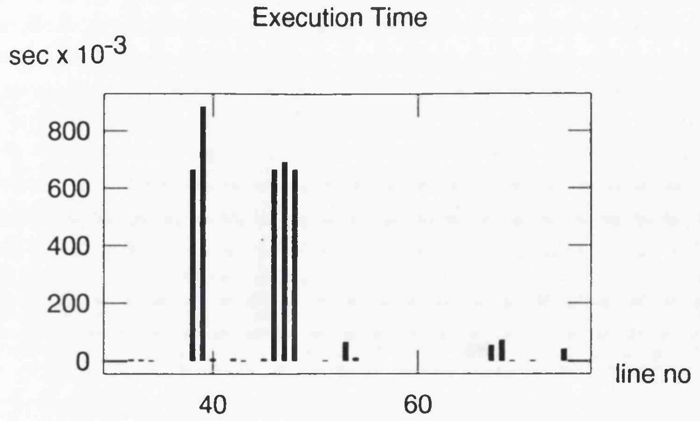


Figure 8.11. Line by Line Computations in SOM

The same pattern recognition problem, used in the Hopfield net simulations is solved using the SOM. A 64 input neurons by 12 output neurons SOM topology is used to cluster 64 dimensional input vectors into 12 classes. The network is trained in an unsupervised fashion, on 12 perfect patterns. After training, noisy input patterns are presented to the network, which are recognised by identifying the nearest output vector.

Data Parallelism in the SOM

CAT projections for the SOM show that little speed-up can be achieved through data parallelism comparing with the Hopfield net with the same problem size and communications speed. This is due to the relatively even distribution of computations on the SOM, and the parallel communications costs which offset the gains of parallel executions. Figure 8.12 shows data parallel CAT projections on the SOM with 64 input 12 output neurons, on a number of parallel VMs with communications speeds between 10 to 40 Mbits/sec. Any significant gain in performance occurs for communication speeds higher than 20 Mbits/sec.

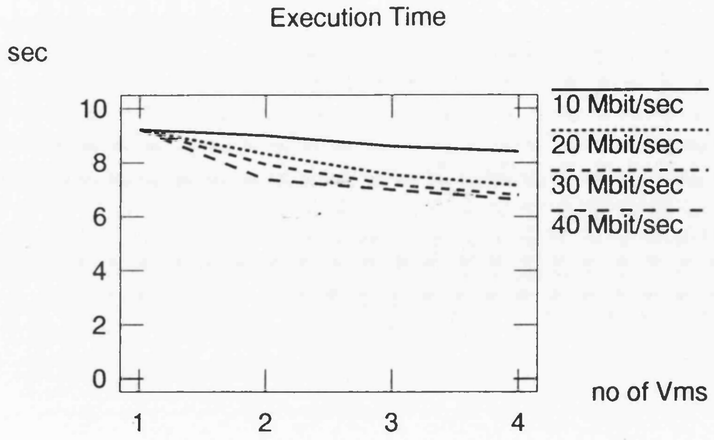


Figure 8.12. Projections on Data Parallel SOM

Task Parallelism on the SOM

CAT results in Appendix D.2 point to the existence of a backward data stream on the weight matrix (SW). This prevents a straightforward instruction pipelining type of parallelism. The reason for the backward data flow is that the weight matrix is read and written at every pattern presentation step. If the changes are accumulated and batches of weight updates are carried out after each epoch, the network can be pipelined through a number of systolic processors, reducing the execution time considerably. The SOM was modified for this purpose, so the weight update changes could be accumulated in a matrix of the same size as the weight matrix, and the batch weight updates can be carried out. This new definition of the SOM was also processed by the CAT (Appendix D.2), and a cutting point on the *MATLIB* listing for an instruction pipeline was found. This partitioning resulted in approximately 30% to 70% load balance, on two parallel processors.

Task Parallel SOM on the SUN LAN

The batch-update version of the SOM was partitioned into two self-contained but interlinked *MATLIB* programs, with their own data definition and data transfer instructions. Two clients and one scheduler programs were compiled and executed on the LAN. This time the scheduler does not take part in neural network related tasks. It only loads data from the file server, distributes the data and waits for data routing instructions. The scheduler, as a passive server is in an infinite loop, and it runs the *MATLIB servis* function. The clients, in turn, use the *post* function which allows them to send data to other VMs, through the server. The server parses these data transmission requests and carries out the orders by rerouting the data.

Figure 8.13 shows the simulation results for a number of SOM configurations executing sequentially, and in parallel on 2 SUN4s. The results confirm that even on general-purpose computing platforms it is possible to improve execution speed by task parallel techniques.

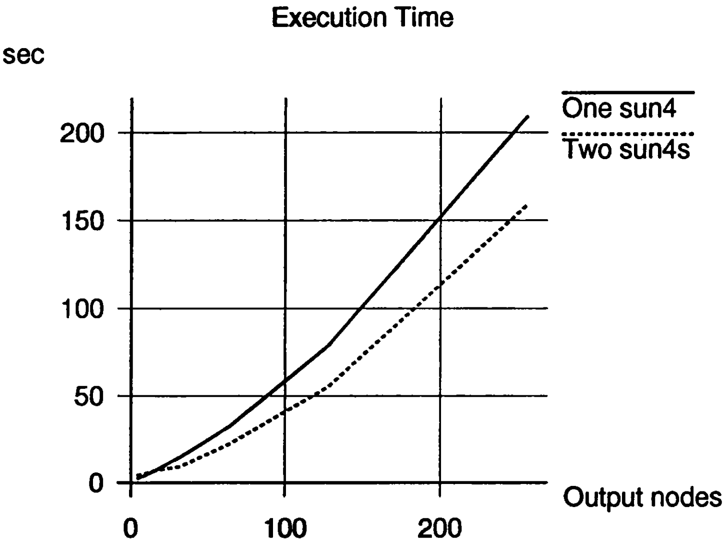


Figure 8.13. Sequential versus Pipelined SOM

To achieve the task parallelism for the SOM a modification had to be made in the weight update procedure of the algorithm. It has been reported that this change can delay or prevent the learning on this model. For this reason, the RMS error change is monitored both in the parallel SOM with batch weight updates and the sequential SOM with single step weight updates on the same pattern recognition problem. Figure 8.14 shows that, although the single step SOM convergence needs less number of iterations for the error to drop to an acceptable level, in the batch update case, the error profile follows the single step SOM error very closely. For this dataset, both methods produce similar results, and the batch weight updates can be used.

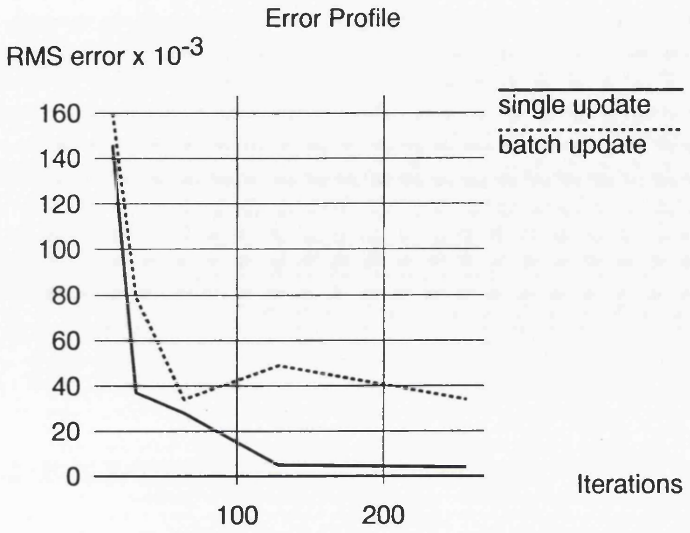


Figure 8.14. RMS Error in Sequential and Parallel SOM

The results show that data parallelism on the SOM can be useful on high speed communications links. Task parallelism is also feasible if the algorithm is modified to carry out batches of weight updates instead of standard single step weight updates.

8.3.3. The Backpropagation Model

The same steps are applied to the Backpropagation model. The *MATLIB* representation of the Backpropagation-with-momentum model is written and processed by CAT to detect possible parallelism and identify computational bottlenecks (Appendix D.3). An even computational profile has emerged, as a result, indicating all operations are computationally demanding (Figure 8.15).

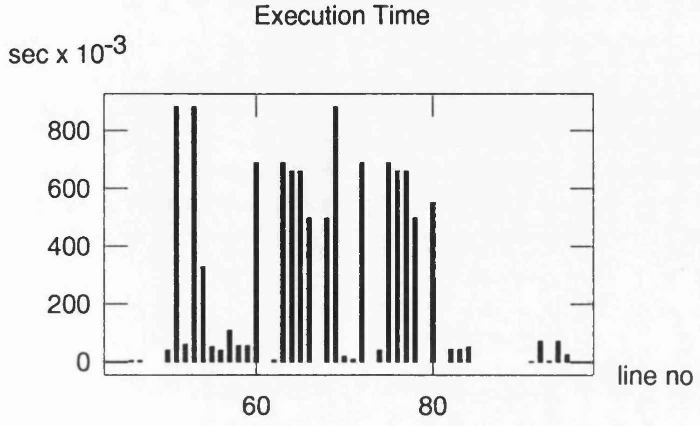


Figure 8.15. Line by Line Computations in Backpropagation

The same pattern recognition problem was used with a three layered Backpropagation network with 64 input, 12 hidden and 64 output neurons. The simulation involved training the network for an auto-associative recall with the 12 base patterns. Once trained, the network was required to generate the same patterns from noisy or incomplete inputs.

CAT projections for a data parallel Backpropagation execution shows that the network can be executed faster, on fast communications links between the parallel VMs (Figure 8.16). Actual parallel simulations on SUN LAN are not implemented as this medium would not meet the communications requirements outlined by the parallel projections. The variable/loop analysis for the Backpropagation-with-momentum model shows a tightly coupled network architecture. Global matrix variables such as the input weights, which are between the input neurons and the hidden layer, and the hidden weights, which are between the hidden layer and the output layer, are all on backward flow data streams. Again, similar to the SOM case, the single step weight update procedure does not allow a profitable task parallelism for the Backpropagation *MATLIB* listing. The only possibility is to modify the algorithm to allow batch updates, and carry out instructions pipelines on a coarse number of processors. Task pipeline simulations was not carried out for the Backpropagation model, as it involves similar steps to the SOM simulation which was previously described.

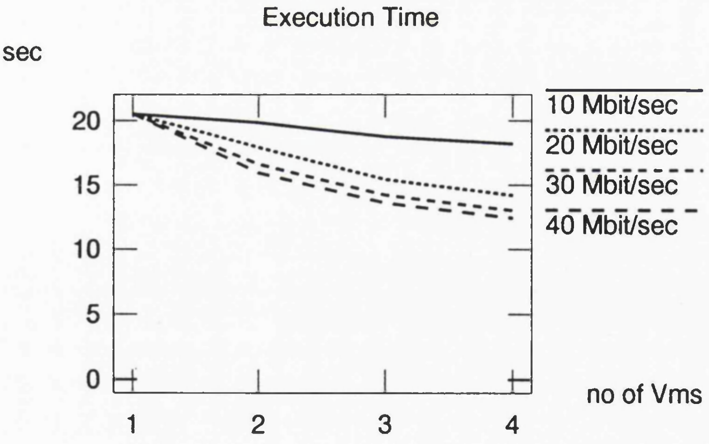


Figure 8.16. Projections on Data Parallel Backpropagation

8.4. Mapping Multiple Neural Networks

In chapter 4, multiple neural network solutions are put forward as powerful ways of enhancing neural network performance in complex pattern recognition problems. Multiple network architectures consist of a number of neural networks as components, with little inter-processor communication requirements, and cooperating or competing to solve a problem. In chapter 7, a Hopfield/Backpropagation architecture is presented, as an example of such systems. In this chapter, two new architectures are put forward, simulated, and mapped onto parallel processors, using *MATLIB* and *NETLIB* functions. The objective of these simulations is to enhance neural network performance through parallelism in terms of achieving efficient parallel executions and designing powerful hybrid systems.

8.4.1. Cooperating SOM/Backpropagation Networks

This simulation demonstrates the cooperation of the SOM and the Backpropagation models, and their parallel execution on parallel processors. The SOM is an unsupervised algorithm, which is used in clustering patterns into a number of classes. The SOM can detect salient features in input patterns, and group them in topologically close nodes on the output grid. The Backpropagation model, on the other hand, operates on pairs of input and target patterns, and builds an internal representation which enables the network to produce nonlinear mappings between inputs and targets. The two networks can be used in cooperation, complementing each other in pattern classification tasks (Figure 8.17). The hybrid architecture involves the SOM acting as a front-end feature detector, filtering inputs to the Backpropagation network which is trained to take appropriate action for the patterns filtered by the SOM. In this case, Backpropagation auto-associates noisy input patterns with the original target patterns. The SOM network receives and clusters noisy input patterns into a number of classes, then, the vectors representing the class centres are used to train the Backpropagation network for an auto-associative recall of the targets. This modular, multi-network configuration has a number of advantages. Firstly, this architecture enhances the strengths of both models; the SOM as a pattern pre-processor, and the Backpropagation as a nonlinear pattern mapping device. Secondly, the noise filtering carried out by the SOM facilitates and speeds up the training of the Backpropagation network.

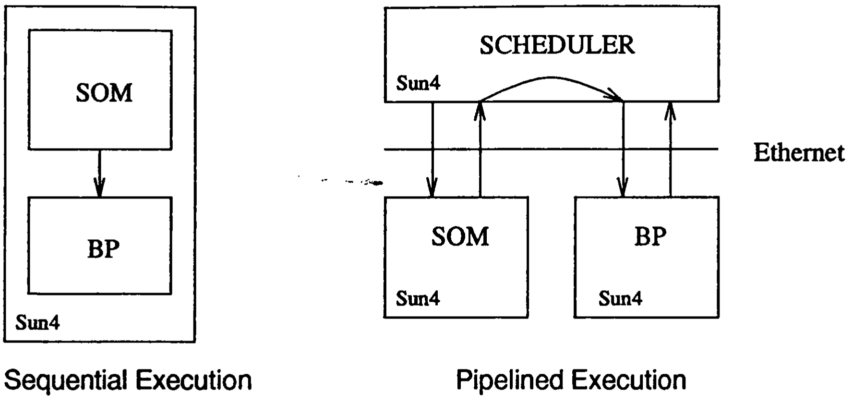


Figure 8.17. Cooperating SOM/Backpropagation Networks

As the application, the same pattern recognition problem has been used. For this application, the SOM has 64 input and 12 output nodes, and the three layered Backpropagation network architecture has 64 input, 12 hidden and 64 output units. A total of 12 patterns is presented to the combined architecture. The following steps have been taken in the simulation: firstly, a flat listing of *MATLIB* definition of the SOM/BP algorithm has been written. This listing is processed by the computational analysis tool, which identifies data-flow paths, variable dependency and computational costs. The analysis reveals a cutting point between the SOM and BP algorithms, suitable for a two-processor pipeline. By dividing the representation into two sections, a pipeline is organised. The partitioned *MATLIB* representation is then parallelised on a 3 processor configuration involving 3 SUN4s. The first SUN workstation is used as the scheduler, which opens a socket, and waits for data transfer requests. The SOM is mapped onto the second SUN which trains on the noisy inputs, transmitting the weight matrix and the winners table to the third SUN station which runs the Backpropagation model, training on the inputs it receives from the SOM and the targets which are local.

The simulation results for the pipelined execution on 3 SUN4s shows an improvement in performance on the sequential execution. The parallel pipelined execution takes only 43 seconds as opposed to the sequential execution resulting in 1:06 min (66 sec). Considering that the first processor is only the server, the results correspond to a speed-up of 1.5 on the two-processor parallel architecture.

Using the CAT results in the identification of the cutting point, the Automatic Parallel Mapper is able to generate automatic parallel pipelined code for this configuration with a near optimum cutting point on the *MATLIB* representation. See Appendix D.4 for CAT and APM results.

8.4.2. Competing Backpropagation Networks

Another method of enhancing neural network performance through parallelism is to implement network level competition. Simulations in this section aim to achieve optimal neural network designs, exploiting computational methods on parallel hardware. In this section, a number of parallel Backpropagation networks are simulated on a number of SUN workstations, competing with each other for a better network topology.

One of the major difficulties in using the Backpropagation model is optimising the network topology and parameters for the network training. These parameters are: the initial set of weights, the learning rate, and the number of nodes in the hidden layer. One way of establishing these parameters is to carry out a number of simulations, and to choose the network configuration with the best results. But this method is too time consuming. A parallel architecture can be used to reduce the time spent in finding an optimised network architecture (Figure 8.18).

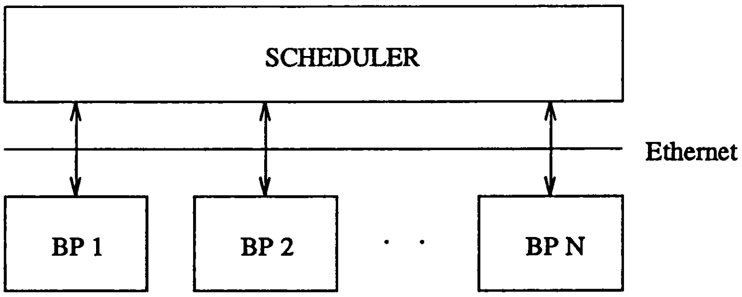


Figure 8.18. Competing Backpropagation Networks

Another difficulty associated with the Backpropagation model is the inherent lack of ability to explain any input-output mapping which the network produces. Input perturbation techniques can be used to identify the most significant input parameter. This method is somehow similar to Monte Carlo simulations. Certain input values are modified, and the outputs are observed, by examining the distribution of inputs and outputs, dependency to the inputs can be established. The same method can be applied to optimise the most significant parameters of the network. Implementation of this method is again too time consuming on sequential architectures, as it involves a number of serial simulations and the comparison of their results. Again, a parallel hardware configuration can be used to obtain results in a shorter time.

In these simulations, the same pattern recognition example as in the previous sections is used. A five processor configuration is designed, involving four

Backpropagation networks, each with a different number of hidden neurons learning the same training dataset. Using the simple *NETLIB* library function *bplearn()*, 4 Backpropagation programs are written, compiled and mapped onto 4 SUN workstations. The scheduler is used as file I/O and a passive server, which monitors error and decides which is the best configuration for the given problem. There is no data dependency and little communication between independent neural network modules. Each Backpropagation network occasionally reports the recall error to the scheduler program which evaluates their performance. *NETLIB* listings of server and client programs are presented Appendix E.

These simulations bring benefits even on general computing platforms such as the SUN LAN used for the simulations. The method of competition is important, as is a practical solution to a complex theoretical problem which interests many neural network researchers in the pursuit of the optimum network design.

8.5. Summary

This chapter presented the simulation results for analysing, partitioning and mapping *MATLIB* representations. A number of simulations are used to confirm CAT results as an approximate computational model of the execution on SUN workstations. CAT is then used to analyse and map *MATLIB* representations of the three neural network models that have been the focus throughout this thesis. Feasibility of data and task parallel executions are investigated for the Hopfield, the SOM and the Backpropagation models, using CAT. Exploiting the results, the three models are partitioned, parallelised and pipelined. Parallel simulations are carried out on a SUN LAN, and simulation results are presented. Finally, multiple neural networks are simulated on a number of parallel processors using *MATLIB* and *NETLIB* functions.

Chapter 9

Assessment

This chapter assesses the thesis work; an investigation of representation and mapping strategies for efficient execution of neural networks on parallel hardware. The thesis consists of a series of analyses, design and implementation work, towards building a general purpose neural computer. Within the context of the research objectives, the analyses, design, implementations and results are assessed, and alternatives are explored.

9.1. Target Review

The goal of this research was to establish a generic mapping strategy for a general purpose neural computing system, which provides a high performance and is flexible, modular, scalable, efficient, and can be automated. Achieving a generic representation was part of this main goal. For this purpose three neural network models and neural network representations have been analysed and compared, a matrix-based library has been put forward; to map matrix-based representations, a computational analysis tool and an automatic parallel mapper have been designed and implemented. Matrix-based representations have been partitioned and parallelised, manually, semi-automatically and automatically. Parallel simulations have been used to assess the performance of the mappings.

The analysis of neural network models aimed to: *(i)* understand neural network models and their computational properties, *(ii)* highlight suitable application domains, *(iii)* explore potential structural parallelism, and *(iv)* search for a generic representation. Three case studies have been used to achieve these aims, involving the Hopfield, the SOM and the Backpropagation models with their appropriate applications.

The analysis of neural network representations aimed to establish a representation which is capable of: *(i)* capturing neural network properties common to most models, *(ii)* exploiting general-purpose parallel computers, and *(iii)* providing generality, flexibility and modularity. For this purpose, function-oriented, object-oriented and vector-oriented representations have been compared, and a matrix-based *C* library, *MATLIB* has been put forward. *NETLIB* also has been developed, which is a neural network library, containing the recall and training functions of the three algorithms.

The main requirements in the design of a generic mapper were ***high performance***, *generality*, *flexibility*, *modularity*, *efficiency*, *scalability* and *automation*. To match these criteria a computational cost based mapping strategy was put forward, and CAT and APM have been designed and implemented to map matrix-based *MATLIB* representations. The same requirements have been applied to the Galatea Mapper which has been designed and developed as part of the Galatea GPNC simulator. Simulations of parallel mappings on the Galatea GPNC involved manual and semi-automatic mappings of *VML* rules, on a number of parallel VMs.

To assess the *performance*, *efficiency* and *scalability*, CAT and APM have been used to partition *MATLIB* definitions of single-domain and multiple neural networks. Parallel simulations have been carried out on a SUN LAN to verify computational analysis projections and demonstrate the use of distributed parallel computer networks for achieving high performance.

In the following sections, the work of this thesis on: the analysis of neural network models, the neural network representations, the execution and mapping strategies, the Computational Analysis Tool, the Galatea Mapper and the Automatic Parallel Mapper, and the performance of the parallel simulations, is assessed.

9.2. Neural Network Analysis

In chapter 4, an analysis of neural networks have been provided. Three most popular models were chosen, and throughout this research, these three models have been used in the analysis of the models and applications, discussions of neural network representations and simulations involving parallel mappings. In the analysis, three appropriate applications have been chosen to highlight computational properties of these models, and to obtain a representative sample of the neural computing field and its applications. The three models have their strengths and weaknesses and they are good at solving different problems.

The Hopfield nets can successfully be used in pattern recognition and data compression tasks. The Hopfield net requires the initial setting of the weight matrix, and it is restrictive in the choice of patterns, as they have to be orthogonal with one another. As the net could not be used with arbitrary patterns for convergence purposes, for realistic applications, real-world patterns must be coded into a set of orthogonal patterns.

The SOM can be used in clustering arbitrary input patterns, according to a distance criterion. The main strengths of this model are; it does not require targets as the input patterns are the targets, it requires very little information about data or problem domain, and the network can continue to learn new datasets as the weights can adapt themselves to the combined dataset. In addition to the parameters relating to the neighbourhood distance and the gain, the most important parameter is the number of nodes in the output grid which determines the number of clusters the network tries to form. Two difficulties are associated with the SOM; dependency to initial conditions, and finding the optimum set of network parameters. Initial random weights influence the resulting distribution of winner nodes for the same dataset, in consecutive runs. This is not desirable as it complicates the interpretation of the results. The second problem involves the setting up of the network parameters. This problem can be eliminated by using network level competition, similar to the competing Backpropagation networks in chapter 8, in order to find the optimum network configuration and parameters.

The Backpropagation model can be used for a wide variety of real-world problems. During the training stage, the network requires input-target pattern pairs. Once the network is trained on a dataset, it can provide nonlinear mappings between inputs and outputs. Because of this, the simulation results are simple to interpret, and this is one of the main reasons for the popularity of this algorithm. A trained network cannot be used for further training on a new dataset, as in that case the weights may be saturated. The correct setting of the learning rate, the momentum term, the choice of the activation function and the selection of the number of hidden units are all important for obtaining correct and robust results.

One outcome of the analysis was that multiple neural networks are expected to be more powerful problem solving domains than single domain systems. These architectures require a generic, modular representation strategy and can be mapped onto general-purpose architectures with relative ease. Chapter 8 provided simulations, exploiting a parallel distributed computer network for neural network level co-operation and competition resulting in clear gains in performance.

Structural analysis of the three models shows that the Hopfield neurons and the SOM output grid can be distributed onto massively parallel neuron-based architectures. In these cases, the interprocessor communications requirements increase polynomially parallel to the number of connections. The Backpropagation model can be partitioned horizontally or vertically, or one-to-one neuron mapping can be considered with a high data traffic on the hidden units. It seems that the massively parallel implementations of these models require fast communications facilities between the processors of the system. This may be seen as shifting the emphasis from high processing speed to high communications speed. The computational optimisation mapping strategy can be also applied to hardware implementations to justify manufacturing.

The analysis showed that the computational requirements for the three models are different in training and recall. In the case of the Hopfield nets, when large matrices are involved, real-time recall requirements can only be met by special-purpose hardware implementations. Both the SOM and Backpropagation models require long training cycles depending on the problem domain, dataset size, dimensionality of the problem, and the hardware platform. The Backpropagation is the most computationally demanding of the three models. Simulations of the three models show that the most demanding computations are multiplications and additions, and the most common aspect of the three models is that they can all be represented by matrix-vector operations. Most of these operations are matrix arithmetic with some exceptions of nonlinear activation functions. This argument was used to abstract neural networks in a computationally more understandable matrix-based representations domain.

9.3. Neural Network Representations

In simulations, neural network data are often processed in the form of arrays and matrices of patterns, weights and outputs. The choice of matrix-based representations is justified as these representations can capture most neural network models at the highest common level. In addition to this, matrix-based representations are suitable for the general-purpose execution strategy which employs vector-matrix based parallel hardware. The main strengths of matrix-based representations are:

- Simplicity and compact representation
- Model independence
- Generality and flexibility

These strengths were demonstrated in the sequential and parallel simulations of the three most popular neural network models.

The main disadvantage of matrix-based neural network representations is that as data are grouped in the form of matrices, the neural network concepts such as layers, clusters, neurons or synapses are not supported. For this reason, mapping these concepts onto neuron-based fine-grained architectures is hampered. An object- or data-oriented approach is more suitable for mapping onto special-purpose neurocomputers.

The two matrix-based representations *VML* and *MATLIB* are suitable for the general-purpose, parallel, accelerator-board-based systems. Matrix-based operations present data to general-purpose SIMD architectures, in a format which is relatively free from data dependencies. These operations can be easily divided and parallelised on SIMD architectures with a good load balance. Considering an element by element multiplication of two matrices (*memu*) on a fine-grained SIMD architecture, the operand matrices' data could be distributed onto the multiplier processors, and a single multiply instruction to all the processors would complete the operation, in a single step. One consideration in the distribution of matrix operations is that some matrix operations are composite. If the operation is divided, the results must be reassembled. An example is the matrix by matrix multiplication (*mmul*) which was parallelised in chapter 8. This operation requires the addition (*madd*) of the parallel submatrix multiplication results. In automatic mapping and parallel code generation, these exceptions can be included in the final system. Alternatively, such composite operations are identified and decomposed into simple, divisible matrix operations.

Another difficulty with the matrix-based representations is the execution of neural operators. As pointed out in chapter 5, some commercial software/hardware provide matrix-based arithmetic operations, but most neural-specific operations still are executed on the sequential host computer. Fine-grained parallel architectures are developed with programmable processors for executing neural activation functions, derivatives etc. Research is in progress in the development of parallel algorithms for matrix operations [115], and the hardware implementations of matrix-based operators.

In this research, two matrix-based representations *VML* and *MATLIB* are used to simulate and map the three neural network models. Both representations proved to be excellent tools which allowed the experimentation and the testing of the models and applications. *VML* suffers from a number of weaknesses:

- Yet another language - The main problem with *VML* is that, it attempts to compete with *C*. As a competitor language it contains similar features to *C* which is unnecessary, as those features are readily available in *C*. As an interpreted language, *VML* has its own parser and interpreter which are planned to be ported onto VMs' local memory space. This is also unnecessary, as nowadays most systems provide their own *C* compilers. When they do not, a cross compiler can be developed which compiles *C* to relevant executable languages. Similar concerns had been raised during the course of the Galatea project [16].

- High-level features - For most neural network simulations a subset of *C* would be sufficient, and *VML* should have consisted of only these features. This would also have facilitated the mapping process, as the calculation of the computational costs would have been simplified. Instead, *VML* includes a large number of statements, with an arbitrary rule hierarchy and a number of loop and control statements. Automatically generated raw *VML* code contains too many *VML* rules, most of which consist of a single line, a call for another rule, reflecting the object-oriented rule hierarchy in the high level language *N*. The rule hierarchy, and the arbitrary control loops complicate the cost analysis for the Galatea Mapper. In fact it is impossible to cost loop statements such as *while* which are resolved only during run-time.

- Low-level features - In addition to accommodating high-level language features, the later version of *VML* (*VML 2.0*) includes the data type in its syntax. As a result, for example, the number of *mmul* statements proliferated to 8, with the introduction of statements like *int8_mmul*, *double32_mmul*, etc. A large number of new statements emerged, each dedicated for an operation with a specific data type. These development make it hard to code in *VML*, as the programmer has to consider the data type at every stage.

MATLIB was originally designed to overcome these weaknesses. As a library, it is basically *C*, containing only the necessary matrix operations sufficient for programming a range of neural network algorithms. It can be extended, using the same conventions and the data structures. Added parallel features make *MATLIB* a matrix-based, parallel, *C*, source library. *MATLIB* has provided a flexible, open, modular environment for neural network programming, ready to be executed on matrix operator parallel hardware.

The use of *NETLIB* functions is even simpler; these functions make neural network programming a matter of calling a *C* function from a program. As future systems are likely to be multiple networks and Hybrid Systems, *NETLIB* functions are a valuable tool for the novice or non-expert. Although it is hidden from the user, the *NETLIB* functions consist of *MATLIB* matrix-based library functions, and these can be executed efficiently on parallel hardware. Both libraries provide a clear, modular and object-oriented means to program neural networks and other similar fine-grained algorithms.

9.4. The Strategy

In this thesis, the neural network execution strategy has been to achieve a high-performance execution for a wide range of models, by exploiting general-purpose parallel hardware platforms. Generality, flexibility and scalability have been other considerations. The main disadvantage of the general-purpose execution strategy is that the general-purpose devices cannot match the level of performance provided by the special-purpose devices [51,70,85,131]. Special-purpose neurocomputers are often application- and algorithm-specific devices and are usually too expensive. The level of performance provided by general-purpose devices can be acceptable for most applications, and can be enhanced through parallelism and cascading. The general-purpose execution strategy shifts the complexity to the software, as it requires flexible representations and efficient mapping strategies which are capable of exploiting general-purpose hardware.

The Virtual Machine concept lies at the centre of the general-purpose execution strategy. This idea is not new - similar ideas have been put forward in the past [23,97]. TRW Mark III, presented in chapter 2, was an early example of a general-purpose, parallel, scalable neurocomputer which pioneered this philosophy in neural computing. The Galatea VM typically consists of a communications unit and an execution unit, each specialised for separate tasks. The communications unit consists of a local memory unit and a CPU, and it is responsible for interfacing with the external environment, controlling the co-processor board and carrying out other calculations which would be too expensive to execute on the board. Execution units are compact, general-purpose neurocomputer, accelerator or co-processor boards. A number of VMs can be connected to a host machine producing a general purpose neural computer. The VMs or general-purpose neurocomputer units are currently under development at Siemens and Philips. After their completion, an assessment of the Galatea GPNC is necessary. The criteria for this assessment would be based on the following requirements, which are also the research objectives for this thesis. They are: high performance, generality, parallelism, flexibility, scalability and modularity.

Siemens and Philips VMs are general-purpose neurocomputers which are expected to yield a high performance, targeting large-size real-world applications. Typical applications include computationally demanding vision tasks, and image recognition and processing. Siemens based VM, SYNAPSE-1, which will be a commercial product, can provide up to 800-1000 MCPS. This is well above the computational requirements for most current neural network applications [6]. Its local storage capacity is also sufficient at 4MBytes, and can be increased by upgrading the local RAM for the communications unit.

Two levels of parallelisms are possible with the VM approach. The execution unit of each VM is a medium- to fine-grained parallel processor array. In addition to this, many VMs can be connected in parallel, providing coarse-grained parallelism. This second level of parallelism is the mapping domain that this thesis work has focused on as part of the development of a general purpose neural computer.

Two radically different mapping/execution philosophies are practised for the execution and mapping of neural networks. The first one exploits the parallel distributed structure of networks; the neural-oriented features such as layers, clusters, neurons and synapses are mapped and executed on parallel distributed hardware. The three neural network case studies in chapter 4 show that these networks favour different types of structural mappings due to the differences in their topological and computational properties. The structural mapping approach is not general or flexible, yet it is simple to understand and can deliver a high performance on massively parallel hardware platforms. Strictly speaking the structural parallelism is data parallelism.

The second mapping approach is based on the high performance execution of the computations involved in neural network simulations. The second approach has been chosen in this thesis, as it is more general, flexible and cost-effective. The mapping strategy based on this computational mapping approach is to develop a mapper as an optimiser. The mapper's main task is then to optimise the use of hardware resources for an efficient execution. The mapper as an optimiser strategy is upgradable. Any optimiser, including genetic algorithms and neural networks can be used to optimise the mapping process. In fact, there have already been attempts to use neural networks as an optimiser in the mapping problem [134].

Central to the optimisation, is the costing of computational load, with two aspects; the processing costs and the communications costs. Most of the parallel mapping efforts have been focused on the computational costing of the sequential and the potentially parallel executions. Naturally, to demonstrate the approach, a linear computational model, and a homogeneous processor architecture have been assumed for simulations on the SUN LAN. The heterogeneous hardwares with nonlinear computational models would be more challenging, although the same principles apply.

The mapping strategy of computational optimisation is generic, and it can also be applied to structural mapping of networks. In that case, it would involve the evaluation of the processing and communications costs for all the objects of the system. Developing a computational optimiser object mapper would be facilitated by object-oriented languages. In fact, the computational look-up tables could be set up as parts of the object classes which are distributed by the mapper. The challenging task for the mapper would be to decide on the level of granularity for partitioning neural network object representations. To achieve this, the potential computational costs of a number of partitionings can be simulated and the execution with the minimum computational cost is chosen. An additional degree of complexity to mapping can be foreseen on the heterogeneous architectures. Then the computational optimisation can last increasingly long in proportion to the granularity of the systems. Neural network or genetic algorithm based optimisers can then replace straightforward computational cost calculations.

9.5. The Implementation

The main challenge, in the implementation of the mapping strategy as a computational optimiser, is to parameterise and estimate the computational costs for potential sequential and parallel executions. The Computational Analysis Tool has been developed for this purpose. Any neural network or other algorithm written by using *MATLIB* functions in the restricted format, can be processed by CAT. The total memory usage is estimated, a line by line analysis of the computations is made, and the possibility of the two types of parallelism - data and task parallelisms - is explored. Then, depending on the number and type of parallel processors available, the data parallel and task parallel communications costs are estimated. Based on the estimates, the Automatic Parallel Mapper then decides whether parallel execution is profitable, and automatically generates data or task parallel *MATLIB* programs.

Automatic code generation schemes, particularly an automatic parallel code generation is very much desired to minimise and altogether eliminate the human effort in programming [32]. The loss of performance during compilations and translations from high-level to low-level representations is one of the major disadvantages of automatic code generation schemes. Writing code in an intermediate-level language is often more efficient than the automatic generation [45]. Where high performance matters, parallel programs can be written manually, or alternatively libraries can be used in parallel programming. *MATLIB* and *NETLIB* libraries were easy to use in manual parallel programming, and throughout this thesis, varying degrees of automation are used in code generation aiming at a fully automatic parallel mapping.

The Galatea Mapper compromises on the automatic generation of *VML* code, by adopting a semi-automatic mapper. At this stage of the project, the computational characteristics of the Virtual Machines are not available. The Galatea project is currently focusing on the optimisation of the execution of the *VML* functions on Siemens and Philips generic boards. Once the generic boards are produced, and all the parameters of the VMs are available, these parameters can be integrated into the semi-automatic mapper, and fully automatic mapping can be realised.

The Automatic Parallel Mapper and the Computational Analysis Tool are strongly linked. CAT does most of the work by costing, analysing, and projecting parallel executions. APM choses one of the options provided by CAT, and based on that, it can generate parallel *MATLIB* definitions automatically from sequential *MATLIB* program listings. CAT simulations in chapter showed the feasibility of the approach by automatically generating *C* code which is compiled and executed. There are two problems with the APM generated code.

Firstly, CAT's data parallel projections search for parallelism on an instruction by instruction basis, ignoring the temporal relationship between the operations. As a result, even constant data matrices are transmitted to the parallel clients for each instance. In fact, constant matrices could be transmitted only once, and stored in the local memories of the VMs throughout the execution. In the parallel simulations of the Hopfield net on the LAN, this approach of transmitting the constant weight matrix only once, is used. The approach resulted in some parallel simulations, on a number of SUNs, running faster than sequential simulations on a single SUN. This behaviour of transmitting constant matrices only once, can be introduced to CAT/APM. In addition to this, a further module could identify temporarily constant variables. These variables stay constant during an inner loop operation until they are written during an outer loop. Again, communication costs can be reduced by transmitting these variables when necessary. There is already a template for these potential developments in the current loop/variable analysis routines in CAT.

Secondly, in task parallel mapping, CAT's variable analysis routine focuses exclusively on the matrix variables, establishing variable dependency links in the form of forward and backward data paths. This is because the transmission costs of these variables are significant in the calculation of the communications costs. In fact, the scalar variables are as important; they can also halt a parallel simulation if they are not received when requested. As a result, the current automatic pipeline mappings may result in incorrect parallel code generation. This problem will be resolved by adding a scalar variable loop analysis routine.

Finally, dynamic or run-time mapping alternatives to compilation-time and static mapping have also been considered but not implemented. This is because dynamic mappings place a heavy demand on communications resources due to the continuous freeze and download operations on the parallel system. For the time being, the initial, static mapping is found sufficient for mapping medium size neural networks.

9.6. Mapping Results and Performance

The mapping results can be assessed in two categories; the Galatea Mapper simulations using *VML*, and parallel mappings of *MATLIB* definitions.

The Galatea Mapper simulations focused on mapping and executing complete networks on general-purpose neurocomputer modules (VMs) of the system. The semi-automatic task mappings showed that the GPNC simulator on the SUN workstations is too slow to gain any speed-up. Mappings of *MATLIB* on the other hand show clear gains for some parallel simulations on the Local Area Network. This discrepancy is due to the slow scheduler on the GPNC, which packs and unpacks messages and transmits data in ASCII. As the purpose of the Galatea GPNC simulator was not to yield high performance, but to provide a developmental, experimental system to the project partners, the performance of the system was acceptable.

The Computational Analysis Tool results on the three models, show that the Hopfield and the SOM models do not scale up, as they contain polynomial relationships in the computational complexity of the algorithms with respect to the network size. This is the case for increasing neuron size for the Hopfield net and increasing output grid size for the SOM. In both cases, computational requirements increase polynomially in proportion to the square of the number of neurons. In these cases, as the network size gets bigger, the computational requirements become unsurmountable. The Backpropagation model shows a linear increase in the computational requirements when the number of neurons is increased linearly. The steepest increase occurs in the increase of the neurons in the hidden layers.

The three neural network models are tightly coupled algorithms with little room for task parallelism. Only by changing the algorithms is it possible to pipeline tasks and achieve a task parallel execution. The same technique of conversion to the batch weight updates, is also used for structural mappings of neural network models. This correspondence confirms that the matrix-based representation adopted in this research captures the same characteristics on a different plane.

The performance of data parallel executions depend on the communications speed and bandwidth of the interconnection architectures. The data parallel execution projections carried out by CAT show above 10 Mbits/sec speeds, data partitioning can be useful on average size networks. The communications bandwidth limits the communications speed when there is a high traffic and large data sizes are transmitted on a bus. The feasibility of the *Future Bus* architecture as the communications medium for a general purpose neural computer has been investigated elsewhere [128] with encouraging simulations results.

Global measures for success on parallel architectures are speed-up and efficiency. Considering a sequential execution lasts T_s, and a parallel execution for the same application on n parallel processors lasts T_p, the speed-up factor is: $S = T_s/T_p$, and the efficiency is: $E = S/n$.

CAT projections in Figure 9.1 show the performance and efficiency for data parallel mapping of a 64-neuron Hopfield net on 2 processors with varying communication speeds. These results show that, the higher the communications speed, the better the speed-up factor and the efficiency.

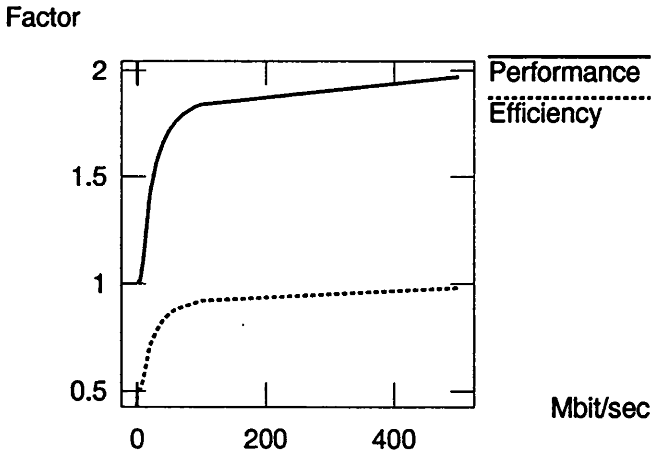


Figure 9.1. Data Parallelism and Communication Speed

A similar relationship is observed between the number of parallel processors and the speed-up factor. However, as seen in Figure 9.2, there is a reverse relationship between the efficiency and the number of parallel processors. These results are also obtained by CAT projections on a 64 neuron Hopfield net, with a constant 10 Mbit/sec communications speed. This indicates that although a faster execution can be achieved, the resources are used inefficiently.

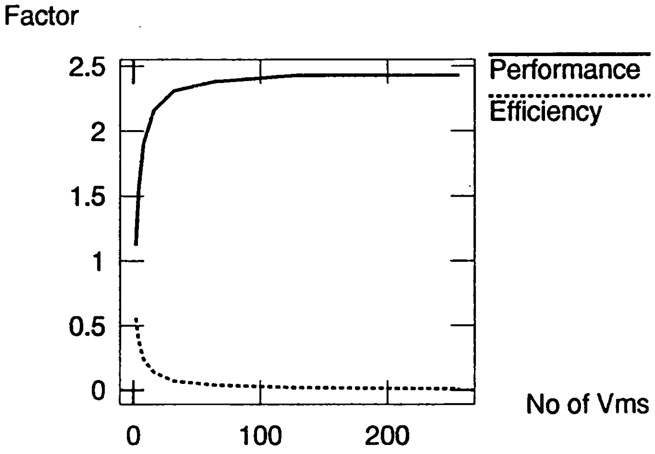


Figure 9.2. Data Parallelism and number of Processors

Another issue in parallel mapping is scalability which is linked to the performance improvement on a parallel system, with respect to the increasing number of processors. The data parallel mapping projections promise a scalable execution particularly for high communications speeds. Task parallel, instruction pipelines, on the other hand, are not scalable as a result of the difficulties in load balancing. Also, CAT pipeline projections showed that finding a cutting point which would result in a pipeline with a good load balance, is not always possible. In any case, scalability is limited by physical interconnection architectures such as buses which can serve only a certain number of parallel processors.

In any case, as shown in chapter 8, current general-purpose, coarse-grained parallel architectures can provide a suitable framework for the integration of different problem solving modules. As the communications requirements are minimal between the independent modules of the system, these architectures can be efficiently used in neural network cooperation and competition for better hybrid algorithm designs, and the genetic optimisation and automatic generation of neural networks.

Chapter 10

Conclusion and Future Work

This final chapter presents the conclusions of this thesis work. The possibility of extending the coarse-grained mapping strategy to fine-grained or heterogeneous architectures is explored.

10.1. Conclusions

The main objective of this thesis was to achieve generic representation and mapping strategies for executing neural networks on parallel hardware. The matrix-based Clibrary, *MATLIB* achieves *generality*, *flexibility* and *modularity*. It captures neural network properties at the most common level, and is suitable for general-purpose parallel computers. The computational cost analysis based mapping strategy is also *general*, *flexible* and, *modular*. It can provide *high performance* and *efficient* execution on parallel machines with high speed communications interfaces. It has been shown that the mapping operation can be automated. The main conclusions of this thesis work are presented on a chapter by chapter basis.

In chapter 4, a comprehensive analysis has been carried out using the three most popular neural network models; the Hopfield, the SOM and the Backpropagation. The following conclusions were reached as a result of the analysis:

- In computer simulations the most common operations are vector-matrix arithmetic operations. A number of neural-network-specific operations exists that can also be represented in a vector-matrix based format. Neural network simulations involve general computing routines such as the file I/O, pattern pre-processing, sorting and graphics interface. These routines can also be abstracted in vector-matrix based representations.

- The three models have their strengths and weaknesses, and popular applications. Combinations of these models in multiple neural network architectures could enhance their performance.

- Structural partitioning techniques applied to a model cannot be generalised to all models.

In chapter 5, function-oriented, object-oriented, and vector-oriented neural network representation techniques have been compared, using a number of simulation examples. The following conclusions have been reached:

- Function-oriented representations focus on the functionality of the algorithms and are suitable for task parallelism, but not suitable for data parallelism.
- Object-oriented representations can capture neural network concepts down to the finest granularity and are are suitable for mapping neural networks onto massively parallel architectures that can match the same level of granularity.
- Vector-oriented representations capture neural network properties at the most common level, and are suitable for exploiting general-purpose parallel computers.

This thesis achieved a generic neural network representation by the design and development of *MATLIB* and *NETLIB* libraries. *MATLIB* functions are general-purpose and encourage programmer to think in a matrix-oriented fashion. They would be suitable for many scientific problems with high dimensionality, and other conventional tasks such as graphics and pattern processing.

In chapter 6, the computational cost analysis based mapping strategy was outlined. It was concluded that the structural mapping techniques are not general or flexible. A computational analysis based mapping strategy is general, flexible and can be upgraded by using modern optimising techniques in future. Combine with the matrix-based representation technique, the mapping strategy can incorporate parallelism in two levels; the matrix-based operations can be pipelined in a MIMD fashion, and each operation can be parallelised in a SIMD or MIMD fashion.

In chapter 7, the design and development of the Galatea Mapper has been presented. The Galatea Mapper distributes a matrix-based common language *VML* onto general-purpose, high performance VMs. The following lessons have been learned:

- Increased complexity in the common language (*VML*), introduce the following difficulties in mapping: *(i)* high-level features such as loop controls that are resolved during the run-time, make compilation-time mapping impossible, *(ii)* low-level features such as hardware-specific instructions should not be in the common representation, and *(iii)* hierarchical structures such as rules in the common representation place unnecessary constraints on the mapping process.

- Increased complexity on the Scheduler slows down simulations to the point that parallelism is useless. Simple, passive scheduling techniques should be adopted for higher performance.

In chapter 8, the results of the CAT projections and parallel simulations have been presented. Parallel, pipelined, single-domain and multiple neural network architectures are used in these simulations. The following conclusions have been reached:

- The three neural network models are tightly coupled algorithms for task parallelism purposes. All three models involve forward and backward data streams in their *MATLIB* representations. Only after modifying their weight update procedures, the SOM and the Backpropagation models can be pipelined through 2 or 3 parallel processors, resulting in a faster parallel execution.

- Data parallelism on an instruction-by-instruction base can produce a speed-up. The amount of speed-up strongly depends on the communications speed. The simulations on the Hopfield model showed that only by parallelising the matrix multiplication a considerable time can be saved. The initial communication costs such as latency must be taken into consideration for fine-grain parallelism.

- Automatic mapping is possible on a small number of parallel processors. CAT and APM can resolve data dependencies and generate parallel and pipelined code, scheduling data transfer operations.

- Current distributed computer networks can be used for speeding up multiple neural network simulations where a number of independent modules compete or co-operate in the solution of a problem.

10.2. Future Work

A comprehensive computer architecture which is capable of serving a hybrid of neural, genetic and rule-based systems is very much desired. Such an architecture would be able to handle a mixture of applications, in a programming environment with multiple representations, and exploit special-purpose, general-purpose and conventional hardware modules, all sharing a message passing communications protocol. Such a system is called a General Purpose Heterogeneous Computer (GPHC) (Figure 10.1). Primary applications for GPHCs would be all fine-grained algorithms such as; Neural Networks, Genetic Algorithms, Virtual Reality systems, Fractal systems, Fluid Dynamics and Finite Element systems.

The programming environment for the GPHC can be realised by linking a number of algorithm libraries, in an open, modular system environment, where users could build their applications by using the library functions as building blocks. The mapping task in this environment, would involve the optimisation of the partitioning and distribution of various representations onto the heterogeneous environment for execution. The computational optimisation mapping strategy can be extended to the heterogeneous system. First, a considerable effort must be put in the computational evaluation of processing and communications costs of the library functions within the multiple representation environment. Special-purpose, fine-grained and general-purpose, coarse-grained, all the modules of the execution environment must be parameterised in the computational cost analysis. Conversion mechanisms between the different libraries must be set up to achieve a load balance in various software and hardware combinations.

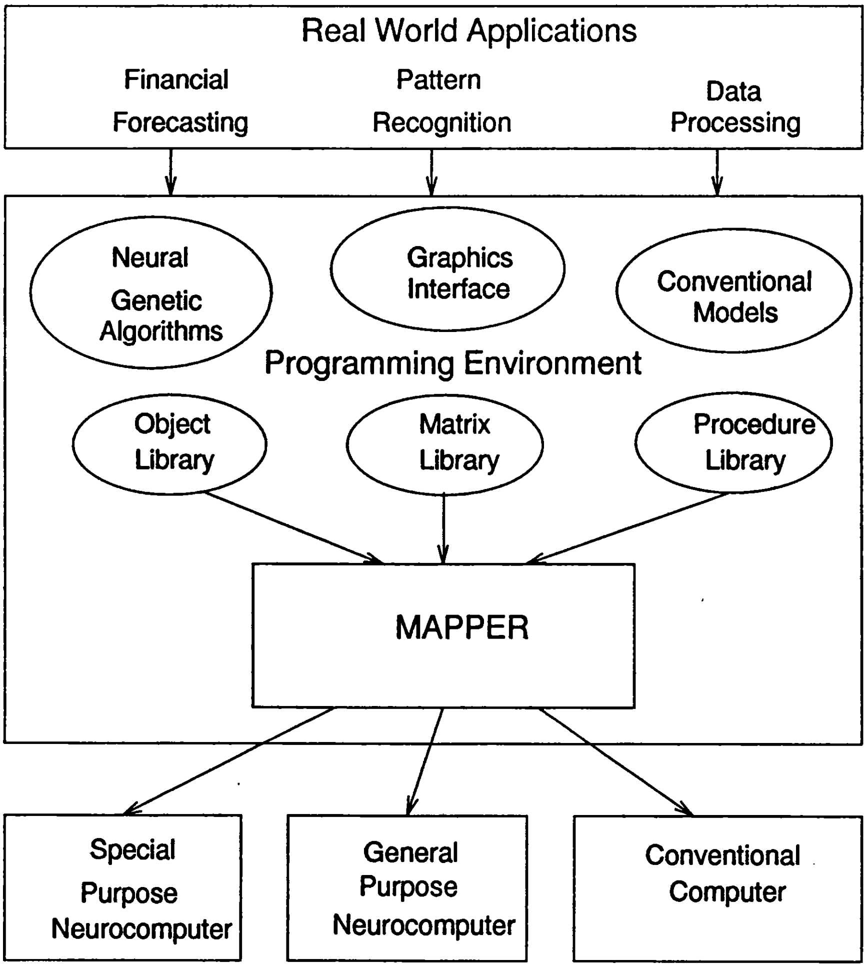


Figure 10.1. General Purpose Heterogeneous Computer

The computational optimisation strategy can also be applied to Silicon Compilers which are becoming increasingly important in the wake of low cost automatic generation of neural ASICs. In this case, the main cost is not only the potential execution time, but also the actual silicon area. Folding algorithms based on optimisers can be used to reduce the final silicon area, through projections and simulations which partition neural network representations and measure the cost in terms of the silicon area.

References

1. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* MIT Press, 1986.

2. *GINNI (Generic Interactive Neural Network Interpreter),* SAIC - Science Applications International Corporation, 1987.

3. Thinking Machines, ''Connection Machine Model CM-2,'' Technical Summary HA87-4, Thinking Machines Corp., April 1987.

4. Nestor, ''Nestor Development System User's Guide,'' Product Information, Nestor Inc., 1988.

5. SAIC, ''DELTA/SIGMA/ANSim, editorial,'' *Neurocomputers,* vol. 2, no. 1, 1988.

6. *DARPA Neural Network Study,* AFCEA International Press, 1988.

7. Meiko Scientific, ''In Sun Computing Surface,'' Product Information, Meiko Scientific Ltd, Bristol, UK, 1989.

8. Adaptive Solutions, ''CNAPS Neurocomputing,'' Product Information, Adaptive Solutions Inc., Beaverton, Oregon, 1991.

9. K. Akingbehin and M. Conrad, ''A Hybrid Architecture for Programmable Computing and Evolutionary Learning,'' *Journal of Parallel and Distributed Computing* , vol. 6, pp. 245-263, 1989.

10. I. Aleksander, W. V. Thomas, and P. A. Bowden, ''WISARD - A radical step forward in image recognition,'' *Sensor Review,* pp. 120-124, July 1984.

11. I. Aleksander, ''Ideal Neurons for Neural Computers,'' in *Parallel Processing in Neural Systems and Computers,* ed. R. Eckmiller, G. Hartmann and G. Hauske, pp. 225-228, Elsevier Science Publishers, 1990.

12. N. M. Allinson, M. J. Johnson, and K. J. Moon, ''Digital Realisation of Self Organising Maps,'' in *Advances in Neural Information Processing Systems 1,* ed. D. S. Touretzky, pp. 728-738, Morgan Kaufmann Publishers, Inc., 1989.

13. J. A. Andersen, ''Cognitive and Psychological Computation with Neural Network Models,'' *IEEE Transactions on Systems, Man, and Cybernetics,* vol. SMC-13, no. 5, pp. 799-814, 1983.

14. B. Angeniol and P. Treleaven, ''PYGMALION: Neural Network Programming & Applications,'' *Proc. ESPRIT 1989 Conference, North Holland, Amsterdam,* 1989.

15. B. Angeniol, ''Pygmalion: ESPRIT II project 2059, Neurocomputing,'' *IEEE Micro,* pp. 28-31 and 99-102, December 1990.

16. B. Angeniol, ''Some Remarks on the VML language,'' Galatea Internal Report, Mimetics, Paris, 1991.

17. J. K. Anlauf, ''SYNAPSE-1 and VML,'' Galatea Internal Report, Siemens, 1991.

18. M. Azema-Barac, M. Hewetson, M. Recce, J. Taylor, P. Treleaven, and M. Vellasco, ''PYGMALION Neural Network Programming Environment,'' *Proceedings of International Neural Network Conference*, Paris, July 9-13, 1990.

19. M. Azema-Barac, ''A Generic Strategy for Mapping Neural Network Models on Transputer-based Machines,'' *Proc. of the Third Int. Conf. on Applications of Transputers*, pp. 768-773, Glasgow, UK, August 28-30, 1991.

20. C. Bahr and D. Hammerstrom, *ANNE (Another Neural Network Emulator),* Intel Scientific Computers, 1987.

21. J. Bailey and D. Hammerstrom, ''Why VLSI implementations of Associative VLCNs Require Connection Multiplexing,'' *Proc. Int. Joint Conf. on Neural Networks*, vol. II, pp. 173-180, San Diego, 1988.

22. N. Baran, ''The Outlook for Pen Computing,'' *Byte*, pp. 159-164, September, 1992.

23. P. Bessiere, A. Chams, and T. Muntean, ''A Virtual Machine Model for Artificial Neural Network Programming,'' *Proceedings of International Neural Networks Conference*, Paris, 1990.

24. U. Bilge and F. Franca, ''Computing with Neural Networks,'' *International Industrial Biotechnology*, vol. 9, no. 4, pp. 17-19, 1989.

25. U. Bilge, ''The Mapper Development,'' Galatea Internal Report REP-S5-M18, UCL CS, 1992.

26. U. Bilge and M. Recce, ''The Self-Organising Map in Clustering Neural Spikes from the rat's Hippocampus,'' UCL CS RN/92/11, 1992.

27. U. Bilge, ''Financial Forecasting using the Self-Organising Map,'' UCL CS RN/92/36, 1992.

28. U. Bilge, ''A Computational Model of the rat's Hippocampus: Navigating through Self-Organising Maps,'' UCL CS RN/92/37, 1992.

29. M. de Bollivier, P. Gallinari, and S. Thiria, ''Cooperation of Neural Nets for Robust Classification,'' *Proc. Int. Joint Conf. on Neural Networks 90*, vol. I, pp. 113-120, San Diego, 1990.

30. M. de Bollivier, P. Gallinari, and S. Thiria, ''Cooperation of Neural Nets and Task Decomposition,'' *Proc. Int. Joint Conf. on Neural Networks 91*, Seattle, Washington US, 1991.

31. H. Bouattour, F. F. Soulie, and E. Viennet, ''Solving the Human Face Recognition Task using Neural Networks,'' *Artificial Neural Networks 2*, pp. 1595-1598, Elseiver Science Publishers, 1992.

32. G. Bricault, ''Juggling Multiple Processors,'' *Byte*, pp. 315-323, May, 1992.

33. G. A. Carpenter and S. Grossberg, ''The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network,'' *IEEE Computer*, March 1988.

34. C. Carter and J. Catlett, ''Assessing Credit Card Applications Using Machine Learning,'' *IEEE Expert*, pp. 71-79, Fall 1987.

35. C. A. Cruz, W. A. Hanson, and J. Y. Tam, ''Neural Network Emulation Hardware Design Considerations,'' *Proc. First Int. Conf. on Neural Networks*, vol. III, pp. 427-434, June 1987.

36. C. Darken and J. Moody, ''Fast Adaptive K-Means Clustering: Some Empirical Results,'' *Proc. Int. Joint Conf. on Neural Networks 90*, vol. II, pp. 233-238, San Diego, 1990.

37. J. Dayhoff, *Neural Network Architectures - An Introduction,* Van Nostrand Reinhold, New York, 1990.

38. D. DeSieno, ''Adding a Conscience to Competitive Learning,'' *Proc. Int. Joint Conf. on Neural Networks 88*, vol. I, pp. 117-124, 1988.

39. R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis,* John Wiley and Sons, New York, 1973.

40. M. Duranton and J. Sirat, ''Learning on VLSI: A General-Purpose Digital Neurochip,'' *Philips Journal of Research*, vol. 45, no. 1, pp. 1-17, 1990.

41. M. Duranton and D. Jorand, ''Hardware Definition of the Generic Board,'' Galatea Internal Report, Philips, Paris, 1991.

42. M. Duranton and P. Friedel, ''Intraboard Communication and Control,'' Galatea Internal Report, Philips, Paris, 1992.

43. S. Dutta and S. Shekhar, ''Bond Rating: A Non-Conservative Application of Neural Networks,'' *Proc. Int. Joint Conf. on Neural Networks 88*, vol. II, pp. 443-450, 1988.

44. H. P. Ernst, B. Mokry, and Z. Schreter, ''A Transputer Based General Simulator for Connectionist Models,'' in *Parallel Processing in Neural Systems and Computers*, ed. R. Eckmiller, G. Hartmann and G. Hauske, pp. 283-286, Elseiver Science Publishers, 1990.

45. G. E. Fagg, P. R. Minchinton, and S. A. Williams, ''The Implementation of Artificial Neural Networks on Parallel Heteregenous Architectures,'' *Artificial Neural Networks, 2*, pp. 1283-1286, Elseiver Science Publishers, 1992.

46. G. Frazier, ''Ariel: A Scalable Multiprocessor for the Simulation of Neural Networks,'' *ACM SIGARCH Computer Architecture News*, vol. 16, no. 1, pp. 107-114, March 1990.

47. K. Fukushima, ''Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition,'' *Neural Networks*, vol. I, pp. 119-130, 1988.

48. K. Fukushima, ''A Neural Network for Visual Pattern Recognition,'' *IEEE Computer*, vol. 21, no. 3, pp. 65-75, March 1988.

49. K. Fukushima, ''Neural Network Models for Visual Pattern Recognition,'' in *Parallel Processing in Neural Systems and Computers*, ed. R. Eckmiller, G. Hartmann and G. Hauske, pp. 351-356, Elseiver Science Publishers, 1990.

50. S. C. J. Garth, ''A Chipset for High Speed Simulation of Neural Network Systems,'' *Proc. of the IEEE First International Conference on Neural Networks*, vol. III, pp. 443-452, June 1987.

51. J. Ghosh and K. Hwang, ''Mapping Neural Networks onto Message-Passing Multicomputers,'' *Journal of Parallel and Distributed Computing*, no. 2, pp. 291-330, Academic Press, 1989.

52. Nigel Goddard, *The Rochester Connectionist Simulator User Manual,* Dept. of Computer Science, University of Rochester, Rochester, 1987.

53. T. Gutschow, *AXON: The Researchers Neural Network Language,* HNC Inc., Hecth-Nielsen Neurocomputers, 1988.

54. D. Hammerstrom, ''A VLSI Architecture for High Performance, Low-Cost, On-chip Learning,'' *Proc. Int. Conf. on Neural Networks 90*, vol. II, pp. 537-544, 1990.

55. W. A. Hanson, C. A. Cruz, and J. Y. Tam, ''CONE - Computational Network Environment,'' *Proc. First Int. Conf. on Neural Networks*, vol. III, pp. 531-538, 1987.

56. R. V. Hanxleden and L. R. Scott, ''Load Balancing on Message Passing Architectures,'' *Journal of Parallel and Distributed Computing* , vol. 13, pp. 312-324, 1991.

57. M. Hardaker, ''Back to the Future,'' *Windows User*, pp. 73-86, July, 1992.

58. S. A. Harp and T. Samed, ''Genetic Optimization of Self-Organizing Feature Maps,'' *Proc. Int. Joint Conf. on Neural Networks 91*, vol. I, pp. 341-346, Seattle, Washington US, 1991.

59. R. Hecht-Nielsen, ''Performance Limits of Optical, Electro-Optical, and Electronic Neurocomputers,'' *Optical and Hybrid Computing SPIE*, vol. 634, pp. 277-306, 1986.

60. R. Hecht-Nielsen, "Counterpropagation Networks," *Proc. First Int. Joint Conf. on Neural Networks*, vol. II, pp. 19-32, 1987.

61. R. Hecht-Nielsen, "Applications of Counterpropagation Networks," *Neural Networks*, vol. 1, pp. 131-139, Pergamon Press, 1988.

62. R. Hecht-Nielsen, "Neurocomputing: picking the human brain," *IEEE SPECTRUM*, vol. 25, no. 3, pp. 36-41, 1988.

63. R. Hecht-Nielsen, *Neurocomputing,* Addison Wesley, San Diego, CA, 1990.

64. W. D. Hillis, *The Connection Machine,* The MIT Press, 1985.

65. G. E. Hinton and T. J. Sejnowski, "Learning and Relearning in Boltzmann Machines," in *Parallel Distributed Processing*, ed. J. L. McClelland, vol. 1, pp. 282-317, MIT Press, 1986.

66. Y. Hirai, "Hardware Implementation of Neural Networks in Japan," *Proc. of the 2nd Int. Conf. on Microelectronics for Neural Networks*, pp. 435-446, 1991.

67. A. Hiraiwa, S. Kuruso, S. Arisawa, and M. Inoue, "A two level pipeline RISC processor array for ANN," *Proc. Int. Joint. Conf. on Neural Networks 90*, vol. II, pp. 137-140, Washington DC, 1990.

68. R. E. Hodges, C.-H. Wu, and C.-J. Wang, "Parallelizing the Self-Organizing Feature Map on Multi-Processor Systems," *Proc. Int. Joint. Conf. on Neural Networks 90*, vol. II, pp. 141-144, Washington DC, 1990.

69. R. M. Holdaway, "Enhancing Supervised Learning Algorithms via Self-Organization," *Proc. Int. Joint. Conf. on Neural Networks 89*, vol. II, pp. 523-530, 1989.

70. M. A. Holler, "VLSI Implementations of Learning and Memory Systems: A Review," in *Advances in Neural Information Processing Systems 3*, ed. D. S. Touretzky, pp. 993-1000, Morgan Kaufmann Publishers, Inc., 1991.

71. M. et al. Homewood, "The IMS T800 Transputer," *IEEE Micro*, vol. 7, no. 5, pp. 10-26, October 1987.

72. J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Nat. Acad. Sci.*, vol. 79, pp. 2554-2558, 1982.

73. J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proc. Nat. Acad. Sci*, vol. 81, pp. 3088-3092, 1984.

74. J. J. Hopfield and D. W. Tank, "Neural" Computation of Decisions in Optimisation Problems," *Bioligical Cybernetics*, vol. 52, pp. 141-152, Springer-Verlag, 1985.

75. T. Hrycej, ''A Modular Architecture for Efficient Learning,'' *Proc. Int. Joint Conf. on Neural Networks 90*, vol. I, pp. 557-562, San Diego, CA, 1990.

76. W. Y. Huang and R. P. Lippmann, ''Neural Net and Traditional Classifiers,'' *Neural Information Processing Systems*, pp. 387-396, American Inst. of Physics, New York, 1988.

77. B. A. Huberman, ''Asynchrony and Concurrency,'' in *Neural Computers*, ed. R. Eckmiller and Ch. v. d. Masburg, pp. 456-465, 1988.

78. A. Iwata, Y. Yoshida, S. Matsuda, Y. Sato, and N. Suzumura, ''An Artificial Neural Network Accelerator using General Purpose 24 Bits Floating Point Digital Signal Processors,'' *Proc. Int. Joint Conf. on Neural Networks 89*, vol. II, pp. 171-175, 1989.

79. D. Jackson and D. Hammerstrom, ''Distributing Back Propagation Networks over the Intel iPSC/860 Hypercube,'' *Proc. Int. Joint Conf. on Neural Networks 91*, Seattle, Washington US, 1991.

80. R. A. Jacobs, M. I. Jordan, and A. G. Barto, ''Task Decomposition through Competition in a modular Connectionist Architecture: The What and Where Vision Tasks,'' COINS TR 90-27, Computer and Information Science, University of Massachusetts at Amherst, USA, March 1990.

81. R. A. Jacobs, M. J. Jordan, S. J. Nowlan, and G. E. Hinton, ''Adaptive Mixture of Local Experts,'' *Neural Computation*, vol. 3, pp. 79-87, 1991.

82. R. F. Jansen, ''The reconstruction of individual spike trains from extracellular multineuron recordings using a neural network emulation program,'' *Journal of Neuroscience Methods*, no. 25, pp. 203-213, Elseiver , 1990.

83. A. Johannet, G. Loheac, L. Personnaz, I. Guyon, and G. Dreyfus, ''A Transputer-Based Neurocomputer,'' *Proc. of the 7th OCCAM Users Group Meeting*, pp. 1-9, 1988.

84. W. P. Jones and J. Hoskins, ''Back-Propagation: A Generalized Delta Learning Rule,'' *Byte*, pp. 155-162, October 1987.

85. H. Kato, H. Yoshizawa, H. Iciki, and K. Askawa, ''A Parallel Neurocomputer Architecture Towards Billion Connection Updates per Second,'' *Proc. Int. Joint Conf. on Neural Networks 90* , vol. II, pp. 47-50, 1990.

86. T. Kimoto, K. Asakawa, M. Yodo, and M. Takeoka, ''Stock Market Prediction System with Modular Neural Networks,'' *Proc. Int. Joint Conf. on Neural Networks 89*, vol. I, pp. 1-6, 1989.

87. J. Kingdon, ''Optimum Weights for Content-Addressable Memory in Neural Nets,'' UCL CS Research Note.

88. T. Kohonen, *Self-Organisation and Associative Memory,* Springer-Verlag, Inc., Berlin, 1984.

89. T. Kohonen, G. Barna, and R. Chrisley, ''Statistical Pattern Recognition with Neural Networks: Benchmarking Studies,'' *Proc. Int. Joint Conf. on Neural Networks 88*, vol. I, pp. 61-68, 1988.

90. T. Kohonen, ''The "Neural" Phonetic Typewriter,'' *IEEE Computer*, pp. 11-22, March 1988.

91. T. Kohonen and K. Makisara, ''The Self-Organizing Feature Maps,'' *Physica Scripta*, vol. 39, pp. 168-172, 1989.

92. T. Kohonen, ''The Self-Organizing Map,'' *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464-1480, 1990.

93. C. Kozakiewicz, T. Ogiso, and N. Miyake, ''Partitioned Neural Network Architecture for Inverse Kinematic calculation of a 6 dof Robot manipulator,'' *Proceedings of International Conference on Neural Networks*, pp. 2001-2006, 1991.

94. O. Kramer and H. Muhlenbein, ''Mapping Strategies in Message-Based Multiprocessor Systems,'' *Parallel Computing*, vol. 9, no. 2, pp. 213-225.

95. S. Y. Kung and J. N. Hwang, ''Parallel Architectures for Artificial Neural Nets,'' *Proc. Int. Joint Conf. on Neural Networks 88*, vol. II, pp. 165-172, San Diego, CA, 1988.

96. H. Lari-Najafi, ''Neural Network Approaches to Automated Knowledge Extraction from Raw Data,'' *PhD Thesis, University of Minnesota*, 1991.

97. J. C. Lawson, A. Chams, and J. Cedex, ''SMART: How to Simulate Huge Networks,'' *Proceedings of International Neural Networks Conference*, Paris, 1990.

98. D. Gilbert Jr. Lee, *Neural Network PC Tools,* Academic Press, 1990.

99. R. P. Lippmann, ''An Introduction to Computing with Neural Nets,'' *IEEE ASSP Magazine*, pp. 4-22, April 1987.

100. J. Loncelle, N. Derycke, and F. F. Soulie, ''Optical Character Recognition and Cooperating Neural Networks techniques,'' *Artificial Neural Networks 2*, pp. 1591-1594, Elseiver Science Publishers, 1992.

101. S. Mackie, H. P. Graf, and D. B. Schwartz, ''Implementations of Neural Network Models in Silicon,'' in *Neural Computers*, ed. R. Eckmiller and Ch. v. d. Masburg, pp. 467-476, 1988.

102. J. R. Mann and S. Gilbert, ''An Analog Self-Organizing Neural Network Chip,'' in *Advances in Neural Information Processing Systems 1*, ed. D. S. Touretzky, pp. 739-747, Morgan Kaufmann Publishers, Inc., 1989.

103. R. Mann and S. Haykin, "A Parallel Implementation of Kohonen Feature Maps on the Warp Systolic Computer," *Proc. Int. Joint. Conf. on Neural Networks 90*, vol. II, pp. 84-87, Washington DC, 1990.

104. E. Marcade, F. Canut, N. Revault, and C. Moulinoux, "N: A Language Dedicated to Neural Algorithms Design," *Thomson-CSF, Esprit II - Pygmalion 2059, Formal HLL Definition, R*, October 1990.

105. S. Margarita, "Recognition of European Car Plates with Modular Neural Networks," *Proceedings of International Neural Networks Conference*, vol. 1, pp. 408-411 , Paris, 1990.

106. T. Matsuoka, H. Hamada, and R. Nakatsu, "Syllable Recognition using Integrated Neural Networks," *Int. Joint Conf. on Neural Networks 90*, vol. I, pp. 251-258, San Diego, 1990.

107. N. Maudit, M. Duranton, J. Gobert, and J.-A. Sirat, "Lneuro 1.0: A Piece of Hardware LEGO for Building Neural Network Systems," *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 414-422, 1992.

108. D. May and R. Shepherd, "The Transputer," in *Neural Computers*, ed. R. Eckmiller and Ch. v. d. Masburg, pp. 478-486, 1988.

109. H. McCartor, "Back Propagation Implementation on the Adaptive Solutions CNAPS Neurocomputer Chip," in *Advances in Neural Information Processing Systems 3*, ed. D. S. Touretzky, pp. 1028-1031, Morgan Kaufmann Publishers, Inc., 1991.

110. W. S. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943. also in Anderson, Rosenfeld (eds.): Neurocomputing

111. R. W. Means and L. Lisenbee, "Extensible Linear Floating Point SIMD Neurocomputer Array Processor," *Int. Joint Conf. on Neural Networks 91*, Seattle, Washington US, 1991.

112. M. Migliore, G. F. Ayala, and S. L. Fornili, "Modeling of Neuronal Systems on Transputer Networks," in *Parallel Processing in Neural Systems and Computers*, ed. R. Eckmiller, G. Hartmann and G. Hauske, pp. 291-294, Elseiver Science Publishers, 1990.

113. M. Miksa, "A Development Tool for Neural Networks Simulations on Transputers," in *Parallel Processing in Neural Systems and Computers*, ed. G. Hauske, pp. 295-298, Elsevier Science Publishers B. V. (North-Holland), 1990.

114. M. Minsky and S. Papert, *Perceptrons*, MIT Press, Cambridge, Mass., 1969.

115. J. J. Modi, in *Parallel Algorithms and Matrix Computation*, Clarendon Press, Oxford, 1988.

116. K. Morse, E. Muenchau, and G. Works, "The SAIC Delta Neurocomputer Architecture," *Proc. 1st Meeting of Int. Neural Network Society*, p. 543, Boston, September 1988.

117. M. E. Nigri, "High Level Synthesis of Neural Network Chips," PhD Thesis, Dept. of Computer Science, University College London, University of London, February 1993.

118. J. Ouali and G. Saucier, "Fast Generation of Neuro-ASICs," *Proc. of the Int. Neural Network Conference 1990*, pp. 563-567, Paris, 1990.

119. J. O'Keefe and L. Nadel, *The Hippocampus as a Cognitive Map,* Oxford University Press, 1978.

120. D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung, "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second," *Int. Joint Conf. on Neural Networks 88*, vol. II, pp. 143-150, San Diego, 1988.

121. D. Pountain and J. Bryan, "All Systems Go," *Byte*, pp. 112-136, August, 1992.

122. U. Ramacher and J. Beichter, "Systolic Architectures for Fast Emulation of Artificial Neural Networks," *Proc. Int. Conf. on Systolic Arrays*, Killarney, Ireland, Prentice Hall, May 1989.

123. U. Ramacher and W. Raab, "Fine-Grain System Architectures for Systolic Emulation of Neural Algorithms," *Int. Conf. on Application Specific Array Processors*, Princeton (USA), September 1990.

124. U. Ramacher, J. Beichter, and N. Bruls, "Architecture of a General-Purpose Neural Signal Processor," *Int. Joint Conf. on Neural Networks 91*, Seattle, Washington US, 1991.

125. M. Recce and P. C. Treleaven, "Parallel Architectures for Neural Computers," in *Neural Computers*, ed. R. Eckmiller and Ch. v. d. Masburg, pp. 487-495, 1988.

126. A. N. Refenes and U. Bilge, "Self-Organising Maps in Pre-Processing Datasets for Decision Support in Histopathalogy," *Proceedings of The North Sea Conference on Biomedical Engineering*, Nov 1990.

127. D. L. Reilly, C. Scofield, C. Elbaum, and L. N. Cooper, "Learning System Architectures Composed of Learning Modules," *IEEE First Int. Joint Conf. on Neural Networks*, vol. II, pp. 495-503, 1987.

128. P. V. Rocha, "A Fully Integrated Neural Computing System," PhD Thesis, Dept. of Computer Science, University College London, University of London, July 1992.

129. M. W. Roth, "Neural-Network technology and its applications," *John Hopkins APL Technical Digest*, vol. 9, no. 3, 1988.

130. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, ''Learning Internal Representations by Error Propagation,'' in *Parallel Distributed Processing*, ed. J. L. McClelland, vol. 1, pp. 318-362, MIT Press, 1986.

131. E. Sackinger, B. E. Boser, J. Bromley, Y LeCun, and L. D. Jackel, ''Application of the ANNA Neural Network Chip to High-Speed Character Recognition,'' *IEEE Transactions on Neural Networks*, vol. 3, no. 3, 1992.

132. M. F. Sarna, P. Gochin, J. Kaltenbach, M. Salganicoff, and G. L. Gerstein,, ''Unsupervised waveform classification for multi-neuron recordings: a real-time, software-based system. II. Performance comparison to other sorters,'' *Journal of Neuroscience Methods*, no. 25, pp. 189-196, Elseiver, 1988.

133. W. Schiffmann and K. Mecklenburg, ''Genetic Generation of Backpropagation Trained Neural Networks,'' in *Parallel Processing in Neural Systems and Computers*, ed. R. Eckmiller, G. Hartmann and G. Hauske, pp. 205-208, Elseiver Science Publishers, 1990.

134. P. Schmid, ''The Mapping Problem: A Neural Network Approach,'' *Proceedings of International Neural Networks Conference*, Paris, 1990.

135. C. L. Seitz, ''Concurrent VLSI Architectures,'' *IEEE Trans. on Computers*, vol. C-33, no. 12, pp. 1247-1264, 1984.

136. H. P. Siemon and A. Ultsch, ''Kohonen Networks on Transputers and Animation,'' *Proc. Int. Joint Conf. on Neural Networks 91*, Seattle, Washington US, 1991.

137. B. Soucek and M. Soucek, *Neural and Massively Parallel Computers / The Sixth Generation,* John Wiley & Sons, 1988.

138. R. Stotzka, R. Hauser, and R. Manner, ''Multiprocessor Simulation of a Self-Organizing Neural Network on NERV,'' in *Parallel Processing in Neural Systems and Computers*, ed. R. Eckmiller, G. Hartmann and G. Hauske, Elsevier Science Publishers, 1990.

139. T. Tanaka, M. Naka, and K. Yoshida, ''Improved Back-Propagation Combined with LVQ,'' *Int. Joint. Conf. on Neural Networks 90*, vol. I, pp. 731-734, Washington DC, 1990.

140. John Taylor, ''VML Specification,'' Galatea Internal Report, UCL CS, 1991.

141. J. B. Theeten, M. Duranton, N. Mauduit, and J. A. Sirat, ''The Lneuro-chip: A Digital VLSI with on-chip Learning Mechanism,'' *Proceedings of International Neural Networks Conference*, Paris, 1990.

142. K. Torkkola, *NEUROCOMPUTERS: from theory to practice, Lecture Notes,* Helsinki University of Technology, Lab. of Inf. and Comp. Science, September 1989.

143. P. Treleaven, M. Pacheco, and M. Vellasco, ''VLSI Architectures for Neural Networks,'' *IEEE Micro*, vol. 9, no. 6, pp. 8-27, December 1989.

144. P. C. Treleaven, ''Neurocomputers,'' *International Journal of Neurocomputing*, vol. 1, no. 1, pp. 4-31, ecn Neurocomputing GmbH, Ismaning, Germany, 1989.

145. P. C. Treleaven, ''Parallel Computing Framework,'' in *Parallel Computers Object-Oriented, Functional, Logic*, ed. P. C. Treleaven, pp. 17-45, John Wiley & Sons, 1990.

146. P. C. Treleaven, ''PYGMALION Neural Network Programming Environment,'' in *Artificial Neural Networks*, ed. T. Kohonen, K. Makisara, O. Simula and J. Kangas, pp. 569-578, Elseiver Science Publishers, 1991.

147. M. M. B. R. Vellasco, ''A VLSI Architecture for Neural Network Chips,'' PhD Thesis, Dept. of Computer Science, University College London, University of London, March 1992.

148. P. D. Wasserman, *Neural Computing - Theory and Practice,* Van Nostrand Reinhold, New York, 1989.

149. T. Watanabe, Y. Sugiyama, T. Kondo, and Y. Kitamura, ''Neural Network Simulation on a Massively Parallel Cellular Array Processor: AAP-2,'' *Int. Joint Conf. on Neural Networks 89*, vol. II, pp. 155-161, 1989.

150. C. Whitby-Strevens, ''Transputers—Past, Present, and Future,'' *IEEE Micro*, vol. 10, no. 6, pp. 16-18 & 76-82, December 1990.

151. M. A. Wilson, U. S. Bhalla, J. D. Uhley, and J. M. Bower , ''Genesis: A System for Simulating Neural Networks,'' in *Advances in Neural Information Processing Systems 1*, ed. D. S. Touretzky, pp. 485-492, Morgan Kaufmann Publishers, Inc..

152. G. A. Works, "The Creation of Delta: A New Concept in ANS Processing,'' *Int. Conf. on Neural Networks 88*, vol. II, pp. 159-164, 1988.

153. M. Yasunaga, N. Masuda, M. Asai, M. Yamada, A. Masaki, and Y. Hirai, ''A Wafer Scale Integration Neural Network Utilizing Completely Digital Circuits,'' *Int. Joint Conf. on Neural Networks*, vol. II, pp. 213-217, Washington DC, June 1989.

154. H. Yoon and J. H. Nang, ''Multilayered Neural Networks on Distributed-Memory Multiprocessors,'' *Proceedings of International Neural Networks Conference*, Paris, 1990.

155. X. Zhang, M. McKenna, J. P. Mesirov, and D. L. Waltz, ''An Efficient Implementation of the Back-Propagation Algorithm on the Connection Machine CM-2,'' in *Advances in Neural Information Processing Systems 2*, ed. D. S. Touretzky, pp. 801-809, Morgan Kaufmann Publishers, Inc., 1990.

Appendix A

APPENDIX A - Neural Network Representations

This appendix presents sections of code from the three representation techniques studied in this thesis.

A.1. The *nC* Data Structure

Below is outline of the *nC* data structure tree for the Pygmalion project.

```
/* ----------------------------------------------------------------------- */
struct SYSTEM   {
                short           n_rules;
                short           n_parameters;
                rule_type       *rules;
                para_type       *parameters;
                int             configs;
                char            **config;
                int             ports;
                port_type       **port;
                int             nets;
                net_type        **net;
        };
/* ----------------------------------------------------------------------- */
struct NET      {
                short           n_rules;
                short           n_parameters;
                rule_type       *rules;
                para_type       *parameters;
                int             layers;
                layer_type      **layer;
        };
/* ----------------------------------------------------------------------- */
struct  LAYER {
                short           n_rules;
                short           n_parameters;
                rule_type       *rules;
                para_type       *parameters;
                int             clusters;
                cluster_type    **cluster;
        };
/* ----------------------------------------------------------------------- */
/* ------------------------------------------------------------------------ */
struct  CLUSTER {
                short           n_rules;
                short           n_parameters;
                rule_type       *rules;
                para_type       *parameters;
                int             neurons;
                int             synapses;
                neuron_type     **neuron;
                synapse_type    **synapse;
        };
/* ------------------------------------------------------------------------ */
struct  NEURON  {
                short           n_rules;
                short           n_parameters;
                rule_type       *rules;
                para_type       *parameters;
                struct NEURON   **input_neuron;
                struct NEURON   **output_neuron;
                int             synapses;
                synapse_type    **synapse;
        };
/* ------------------------------------------------------------------------ */
struct  SYNAPSE {
                short           n_rules;
                short           n_parameters;
                rule_type       *rules;
                para_type       *parameters;
        };
/* ------------------------------------------------------------------------ */
```

A.2. The Hopfield Recall Rule in *C*

Below is the recall rule for the Hopfield simulation in *C*.

```
/* ------------------------------------------------------------------------ */
int recall(c_pat)
int     c_pat;
{
        int     i, iter;
        double  rms;

        for (i=0; i<NEURONS; i++) {
                old_out[i] = pats[c_pat][i];
                out[i] = pats[c_pat][i];
        }
        iter  = 0;
        while (iter < LIMIT) {
                for (i=0; i<NEURONS; i++) {
                        dot_product(i);
                        threshold(i);
                }
                rms = 0.0;
                for (i=0; i<NEURONS; i++) {
                        rms += (out[i] - old_out[i])*(out[i] - old_out[i]);
                }
                if (rms == 0.0) {
                        printf("pattern %d Converged Iteration=%d0, c_pat, iter);
                        break;
                }
                iter++;
                for (i=0; i<NEURONS; i++) {
                        old_out[i] = out[i];
                }
        }
        return(NOTOK);
}
/* ------------------------------------------------------------------------ */
int dot_product(i)
int i;
{
        int j;

        for (j=0; j<NEURONS; j++) {
                out[i] += old_out[j] * weight[i][j];
        }
}
/* ------------------------------------------------------------------------ */
int threshold(i)
int i;
{
        out[i] = (double) tanh( out[i] );
}
/* ------------------------------------------------------------------------ */
```

A.3. The Decrement Distance Function in *nC*

Below is the Decrement_distance rule for the SOM simulation in *nC*.

```
/* ---------------------------------------------------------------- */
int Decrement_distance() /* reduce distance and gain */
{
    if (sys->net[c_net]->parameters[ NET_P_distance ].parameter.value.f <=
        sys->net[c_net]->parameters[ NET_P_distance_finish ].parameter.value.f ) {
        return ( TERM );
    }

    sys->net[c_net]->parameters[ NET_P_gain ].parameter.value.f -=
        sys->net[c_net]->parameters[ NET_P_gain_step ].parameter.value.f;

    if ( sys->net[c_net]->parameters[ NET_P_gain_step ].parameter.value.f > 0.0  ) {
        if ( sys->net[c_net]->parameters[ NET_P_gain ].parameter.value.f <
              sys->net[c_net]->parameters[ NET_P_gain_finish ].parameter.value.f ) {
                sys->net[c_net]->parameters[ NET_P_gain ].parameter.value.f =
                sys->net[c_net]->parameters[ NET_P_gain_finish ].parameter.value.f;
              }
    }
    else {
        if ( sys->net[c_net]->parameters[ NET_P_gain ].parameter.value.f >
            sys->net[c_net]->parameters[ NET_P_gain_finish ].parameter.value.f ) {
                sys->net[c_net]->parameters[ NET_P_gain ].parameter.value.f =
                sys->net[c_net]->parameters[ NET_P_gain_finish ].parameter.value.f;
            }
    }
    sys->net[c_net]->parameters[ NET_P_distance ].parameter.value.f -=
    sys->net[c_net]->parameters[ NET_P_distance_step ].parameter.value.f;

    if ( sys->net[c_net]->parameters[ NET_P_distance ].parameter.value.f < 0.0 ) {
        sys->net[c_net]->parameters[ NET_P_distance ].parameter.value.f = 0.0;
    }
    return ( 0 );
}
/* ---------------------------------------------------------------- */
```

A.4. The Recall Rule for the Backpropagation in *VML*

Below is the data definition and Recall rules for the Backpropagatiom model in *VML*.

```
#               name    type    initial value
define_scalar:  "X",    0,      4               # input layer
define_scalar:  "Y",    0,      2               # hidden layer
define_scalar:  "Z",    0,      4               # output layer
define_scalar:  "P",    0,      4               # no. of training patterns
define_scalar:  "T",    0,      0.25            # tolerance
define_scalar:  "N",    0,      0.02            # learning rate

#               name    type    m       n
define_matrix:  "S0",   0,      P,      X       # Input states - training patterns
define_matrix:  "S1",   0,      1,      Y       # Hidden states row vector
define_matrix:  "S2",   0,      1,      Z       # Output states row vector
define_matrix:  "W1",   0,      X,      Y       # Weights between input and hidden lay.
define_matrix:  "W2",   0,      Y,      Z       # Weights between hidden and output lay.
define_matrix:  "E1",   0,      1,      Y       # Errors in hidden layer
define_matrix:  "E2",   0,      1,      Z       # Errors in output layer
define_matrix:  "A1",   0,      1,      Y       # Accumulator in hidden layer
define_matrix:  "A2",   0,      1,      Z       # Accumulator in output layer
define_matrix:  "PE",   0,      P,      Z       # individual errors in all patterns
#####################################################################################
define_rule:    Recall( p )
                A1      = mu ( S0[p,*], W1 )    # [p,*] refers to pth row vector
                S1      = afm( tanh, A1 )       # activation function for hidden layer
                A2      = mu ( S1, W2 )         # multiply hidden states
                S2      = afm( tanh, A2 )       # activation function for output layer
                E2      = es ( S0[p,*], S2 )    # subtract outputs from input pattern
                PE[p,*] = cp ( E2 )             # copy to PE
```

Appendix B

APPENDIX B - MATLIB Functions

This appendix presents a complete list of the MATLIB functions. The functions are grouped into four categories; data, arithmetic, neural and communications operators.

Data Operators

Data operators relate to the memory allocation, memory management and file I/O operations. The syntax and the arguments for the *MATLIB* data operators are as follows:

matdef (matrix_name, "matrix_name", rows, columns);

Matrix definition; for setting up matrix data structures and memory allocation.

mcpy (destination, source);

Matrix copy; copy froms source to destination, both arguments are pointers to matrix structures.

vcpy (destination, source, index1, index2);

Matrix row copy; arguments are; two matrix pointers and two indices, index1 refers to the destination matrix and index2 refers to the source.

sval (scalar_addr, source, row, column);

Set scalar value from matrix element; the pointer of the double precision scalar value is passed to the function, the element of the source with the row and column indices is assigned to the scalar.

mset (destination, scalar, row, column);

Matrix element set; the row, column addressed element of the matrix is set to scalar value.

msal (destination, scalar);

Matrix all elements set; all matrix elements are set to the scalar value.

mtra (destination, source);

Matrix transpose; source transposed and placed into destination.

matsv (filename, source);

Matrix save; save source matrix into filename.

matld (destination, filename);

Matrix load; read matrix from filename.

matsh (filename, source, row, threshold, col);

Output matrix source to filename or the standard output, 'row' indexed pattern from the source matrix, displayed with line breaks at every 'col' for a matrix display format with values above the threshold are shown as 'X', below the threshold as '.', if zero ' '.

mout (source);

Matrix output; source matrix values are dumped to the standard output for debugging purposes.

mran (destination, min, max)

Matrix randomise; generate random values between min and max, write to destination.

Arithmetic Operators

Arithmetic operators relate to the matrix arithmetic operations such as addition, multiplication etc.

mmul (destination, source1, source2);

Matrix multiplication; source1 and source2 are multiplied and the result is placed into destination. All arguments are pointers to matrix structures.

mtrm (destination, source1, source2);

Matrix multiplication; the same as above with the second source matrix transposed. This saves the execution of transposition as a separate operation.

madd (destination, source1, source2);

Matrix additions; all arguments are pointers to matrix structures.

msub (destination, source1, source2);

Matrix subtraction; all arguments are pointers to matrix structures.

memu (destination, source1, source2);

Matrix element by element multiplication, all arguments are pointers to matrix structures.

vadd (destination, source1, source2, index1, index2, index3);

Matrix row addition; three matrices, and three respective row indices.

vsub (destination, source1, source2, index1, index2, index3);

Matrix row subtraction; three matrices, and three respective row indices.

vemu (destination, source1, source2, index1, index2, index3);

Matrix row element by element multiplication, three matrices, and three respective row indices.

mscm (destination, source, scalar);

Matrix by scalar multiplication, all elements of the source is multiplied by the scalar and the result is put into destination.

mnor (destination, source)

Normalise all the rows of the source matrix and put the resulting vectors into the destination matrix.

mavg (scalar_addr, source)

Matrix average; calculate the mean average of the source matrix.

mmax (scalar_addr, source, row_addr, col_addr)

Matrix maximum; find the maximum element of the source matrix, and return its value in scalar, and its position in row and column.

mmin (scalar_addr, source, row_addr, col_addr)

Matrix minimum; find the minimum element of the source matrix, and return its value in scalar, and its position in row and column.

mabs (destination, source)

Matrix absolute; take the absolute values of a source matrix put to destination.

Neural Operators

Neural operators are neural network specific functions which are applied to a matrix. They can also be implemented in parallel, yet they require an enhanced level of complexity on parallel processors.

mtan (destination, source)

Apply tangent hyperbolic function to all elements of the source matrix, and put the result into the destination.

dtan (destination, source)

Apply derivative of the tangent hyperbolic to the source matrix.

msig (destination, source)

Apply sigmoid function to the source matrix.

dsig (destination, source)

Apply derivative of the sigmoid function to the source matrix.

mrms (scalar_addr, source)

Matrix root mean squared; calculate the rms of the matrix pass it to the scalar, through its pointer.

mdis (destination, dimensions)

Matrix distance; this function is specific to the SOM, it calculates distance values between the nodes of the output grid with a given dimensionality. The function is called once at the beginning of the training to set up the distance values. Once set, another function *mlat* handles the neighbourhood decay and lateral weight update.

mlat (destination, source, distance)

Matrix lateral; this function takes the source matrix and a given distance and reforms the lateral weight matrix for this distance. Again it is specific to the SOM, and used to carry out the lateral weight update.

Communications Operators

These operators exploit TCP/IP sockets for data transfer between a number of SUN workstations on a Local Area Network. The functions are generic, and their equivalents can be easily implemented on multi-computer parallel systems.

put_str (socket_id, string)

Transmit string through the open socket with socket_id.

put_int (socket_id, int_scalar)

Transmit integer scalar value through the open socket with socket_id.

put_dbl (socket_id, double_scalar)

Transmit double scalar value through the open socket with socket_id.

put_mat (socket_id, matrix_name)

Transmit matrix with all double precision values through the open socket with socket_id.

put_rows (socket_id, matrix_name, index1, index2)

Transmit matrix elements between the row indices; index1 and index2.

put_cols (socket_id, matrix_name, index1, index2)

Transmit matrix elements between the column indices; index1 and index2. Together with *put_rows* this submatrix data transfer function is used in data partitioning and data parallel executions.

put_val (socket_id, double_scalar, every)

Transmit double precision scalars once at 'every' iteration. This is for debugging purposes for graphics or screen display of the network performance.

get_str (socket_id, string)

Receive string through the open socket with socket_id.

get_int (socket_id, int_scalar)

Receive integer scalar value through the open socket with socket_id.

get_dbl (socket_id, double_scalar)

Receive double scalar value through the open socket with socket_id.

get_mat (socket_id, matrix_name)

Receive matrix with all double precision values through the open socket with socket_id.

get_val (socket_id, double_scalar, every)

Receive double precision scalars once at 'every' iteration.

Appendix C

APPENDIX C - MATLIB Listings of the three Models

This appendix presents the listings of the Hopfield, the SOM and the Backpropagation MATLIB simulations.

C.1. The Hopfield *MATLIB* Listing

Below is the *MATLIB* listing of the Hopfield net simulation. The net has 64 neurons, and it processes 12 patterns allowing only 4 iterations per pattern for convergence.

```
#include "uv.h"
int    NPATS = 12;
int    INSIZE = 64
int    JL = 4;
char   *weigfil="weig";
char   *infile="r.dat";
double  rms;
int    i, j;
mt_type      *S0, *S1, *ST, *HW, *ST0, *SD;
main()
{
      matdef(&S0, "S0", NPATS, INSIZE);
      matdef(&S1, "S1", NPATS, INSIZE);
      matdef(&HW, "HW", INSIZE,INSIZE);
      matdef(&ST0, "ST0", 1,   INSIZE);
      matdef(&ST, "ST", 1,     INSIZE);
      matdef(&SD, "SD", 1,     INSIZE);

      matld(S0, infile);
      matld(HW, weigfil);
      for (i=0; i<NPATS; i++) {
            printf ("Recalling %d0,i);
            vcpy(ST0, S0, 0, i);
            for (j=0; j<JL; j++){
                  mmul(ST, ST0, HW);
                  mtan(ST, ST);
                  msub(SD, ST, ST0);
                  mrms(&rms, SD);
                  if ( rms == 0.0 ) {
                        break;
                  }
                  mcpy(ST0, ST);
            }
            matsh("", S0, i, 0.0, 8);
            printf ("rms = %f0,rms);
            vcpy(S1, ST, i, 0, 1);
            matsh("", S1, i, 0.0, 8);
      }
}
```

C.2. The SOM *MATLIB* Listing

Below is the *MATLIB* listing of the SOM simulation. The network topology comprises 64 input neurons and 12 nodes in the output grid. 12 patterns are presented to the network, and during the training maximum 30 iterations are allowed.

```
#include "uv.h"
int    NP = 12;
int    IN = 64;
int    OU = 12;
int    JL = 30;
char   *infile="h.dat";
char   *weigfil="soweig";
double  dist = 2.0;
double  dstp = 0.4;
double  dend = 0.1;
double  gain = 0.9;
double  gstp = 0.1;
double  gend = 0.1;
double  rms, tmp, avg;
int    i, j, k, row, col;
mt_type *S0, *S1, *SW, *LW, *LD, *ST, *SC, *SE;
main()
{
      matdef(&S0, "S0", NP,    IN);
      matdef(&S1, "S1", NP,    IN);
      matdef(&SE, "SE", 1,     NP);
      matdef(&SW, "SW", OU,    IN);
      matdef(&LW, "LW", OU,    OU);
      matdef(&LD, "LD", OU,    OU);
      matdef(&ST, "ST", 1,     IN);
      matdef(&SC, "SC", 1,     OU);

      matld(S0, infile);
      mran( SW, 0.0, 0.05);
      mdis( LD, 2);
      mlat( LW, LD, dist );
      mscm( LW, LW, gain);
      for (i=0; i<JL; i++) {
            for (j=0; j<NP; j++) {
                  for (k=0; k<OU; k++) {
                        vsub(ST, S0, SW, 0, j, k);
                        mrms(&rms, ST);
                        mset(SC, rms, 0, k);
                  }
                  mmin(&rms, SC, &row, &col);
                  mset(SE, rms, 0, j);
                  for (k=0; k<OU; k++) {
                        sval(&tmp, LW, k, col);
                        vsub(ST, S0, SW, 0, j, k);
                        mscm(ST, ST, tmp);
                        vadd(SW, SW, ST, k, k, 0);
                  }
            }
            mavg(&avg, SE);
            printf ("%d: avg rms = %f0, i, avg);
            mlat( LW, LD, dist );
            mscm(LW, LW, gain);
            dist = dist - dstp;
            if ( dist < dend ) {
                  dist = dend;
            }
```

```
          gain = gain - gstp;
          if ( gain < gend ) {
                gain = gend;
          }
     }
     for (j=0; j<NP; j++) {
          for (k=0; k<OU; k++) {
                vsub(ST, S0, SW, 0, j, k);
                mrms(&rms, ST);
                mset(SC, rms, 0, k);
          }
          mmin(&rms, SC, &row, &col);
          printf ("pat %d min rms = %f winner=%d0, j, rms, col);
          matsh("", S0, j, 0.0, 8);
          vcpy(S1, SW, j, col);
          matsh("", S1, j, 0.0, 8);
     }
     matsv(weigfil, SW);
}
/*--------------------------------------------------------------*/
```

C.3. The Backpropagation *MATLIB* Listing

Below is the *MATLIB* listing of the Backpropagation simulation. The network topology comprises 64 input, 12 hidden and 64 output neurons. A total of 12 patterns are used as target patterns for a variety of noisy inputs. A maximum of 23 iterations are allowed during the training.

```
#include "uv.h"
int    NP = 12;
int    IN = 64;
int    HID = 12;
int    OUT = 64;
int    JL = 23;
char   *infile="r.dat";
char   *tarfile="h.dat";
char   *oweig="oweig";
char   *hweig="hweig";
double  gain = 0.003;
double  momt = 0.5;
double tol = 0.4;
double tmp, avg, max;
int    i, j, p, row, col;
mt_type      *SI, *Si, *ST, *Sr, *Sh, *PE;
mt_type      *W1, *W2, *WT, *A1, *A2, *ER;
mt_type *M1, *M2, *E2, *D1, *D2, *SR;
main()
{
      matdef(&SI,  "SI", NP,   IN);
      matdef(&Si,  "Si", 1,    IN);
      matdef(&ST,  "ST", NP,   OUT);
      matdef(&SR,  "SR", NP,   OUT);
      matdef(&Sr,  "Sr", 1,    OUT);
      matdef(&Sh,  "Sh", 1,    HID);
      matdef(&PE,  "PE", NP,   OUT);
      matdef(&ER,  "ER", 1,    OUT);
      matdef(&W1,  "W1", IN,   HID);
      matdef(&M1,  "M1", IN,   HID);
      matdef(&W2,  "W2", HID,  OUT);
      matdef(&M2,  "M2", HID,  OUT);
      matdef(&A1,  "A1", 1,    HID);
      matdef(&D1,  "D1", 1,    HID);
      matdef(&E2,  "E2", 1,    OUT);
      matdef(&A2,  "A2", 1,    OUT);
      matdef(&D2,  "D2", 1,    OUT);
      matld(SI, infile);
      matld(ST, tarfile);
      mran( W1, -0.05, 0.05);
      mran( W2, -0.05, 0.05);
      for (j=0; j<JL; j++) {
            for (p=0; p<NP; p++) {
                  vcpy(Si, SI, 0, p);
                  mmul(A1, Si, W1);
                  mtan(Sh, A1);
                  mmul(A2, Sh, W2);
                  mtan(Sr, A2);
                  vsub(E2, ST, Sr, 0, p, 0);
                  vcpy(PE, E2, p, 0);
                  dtan(A2, Sr);
                  memu(D2, E2, A2);
                  mscm(D2, D2, gain);
                  mscm(M2, M2, momt);
                  for (i=0; i<HID; i++) {
                        sval(&tmp, Sh, 0, i);
                        mscm(A2, D2, tmp);
                        vadd(A2, A2, M2, 0, 0, i);
                        vadd(W2, W2, A2, i, i, 0);
                        vcpy(M2, A2, i, 0);
                  }
                  mtrm(D1, D2, W2);
                  dtan(A1, Sh);
                  memu(D1, D1, A1);
                  mscm(M1, M1, momt);
                  for (i=0; i<IN; i++) {
                        sval(&tmp, Si, 0, i);
                        mscm(A1, D1, tmp);
                        vadd(A1, A1, M1, 0, 0, i);
                        vadd(W1, W1, A1, i, i, 0);
                        vcpy(M1, A1, i, 0);
                  }
                  mabs(M1, M1);
            }
               mabs(PE, PE);
               mmax(&max, PE, &row, &col);
               mavg(&avg, PE);
               printf("%4d: max=%f avg=%f tol=%f0, j, max, avg, tol);
               if ( max < tol ) {
                       break;
               }
        }
        for (p=0; p<NP; p++) {
               vcpy(Si, SI, 0, p);
               mmul(A1, Si, W1);
               mtan(Sh, A1);
               mmul(A2, Sh, W2);
               mtan(Sr, A2);
               matsh("", Si, 0, 0.0, 8);
               matsh("", Sr, 0, 0.0, 8);
        }
}
/*-------------------------------------------------------------*/
```

Appendix D

APPENDIX D - CAT and APM Results

This appendix presents the CAT and APM results on the Hopfield, the SOM and the Backpropagation MATLIB listings and a multiple network architecture SOM/Backpropagation model.

D.1. The Hopfield Net

In this simulation a 64 neuron Hopfield net is analysed by CAT/APM for potential data parallelism and task parallelism. First the memory use and sequential execution costs on a SUN4 station are shown, then the possibility of data parallel execution on 2 VMs is searched and finally the feasibility of task parallelism on 2 VMs is investigated. Below is the CAT results for the *MATLIB* listing of the Hopfield net.

```
   0:    S0[  12 ][  64 ] ->  768
   1:    S1[  12 ][  64 ] ->  768
   2:    HW[  64 ][  64 ] -> 4096
   3:   ST0[   1 ][  64 ] ->   64
   4:    ST[   1 ][  64 ] ->   64
   5:    SD[   1 ][  64 ] ->   64
TOTAL MATRIX MEMORY=   5824 Elements ->  46592 Bytes

line func    repeat size unit   lcost        comp       total
 24: vcpy     12      64  4.5  0.000289     0.003      0.003
 26: mmul     48    4096  8.0  0.032768     1.573      1.576
 27: mtan     48      64 36.0  0.002307     0.111      1.687
 28: msub     48      64  6.0  0.000384     0.018      1.705
 29: mrms     48      64  8.0  0.000512     0.025      1.730
 33: mcpy     48      64  4.5  0.000289     0.014      1.744
 37: vcpy     12      64  4.5  0.000289     0.003      1.747
TOTAL COMPUTATIONAL COST=      1.747 seconds
```

In the results above a line by line analysis is presented; for example, in line 24 a *vcpy* operation takes place which will be repeated 12 times as a result of the loops, the data size copied is 64, the operation unit cost is 4.5 micro seconds. As a result the line cost is 0.000289 seconds, the computational cost for the line is 0.003 seconds and finally total accumulative cost for the listing is 0.003 seconds.

Below the data parallel projections are shown. Each line is evaluated separately. CAT/APM, in this case presumes 10 Mbit/sec communications speed and 2 parallel

VMs. For example, for the *mmul* operation, the sequential cost is calculated as 0.032768 seconds, and the data parallel execution, including the data distribution and reassembly costs would be 0.029798 seconds. This would be profitable, and $$$ sign indicates the partitioning decision in this case. For other operations the same steps are repeated, and overall the parallel execution is estimated to last 1.56 seconds. In comparison with 1.747 seconds sequential execution cost this result correspond to a 1.12 times speed-up on 2 processors with 0,56 efficiency factor.

```
func    sequential  parallel    total
 vcpy   0.000289    0.002705   0.003471
 mmul   0.032768    0.029798   1.433795 $$$
 mtan   0.002307    0.001460   1.503898 $$$
 msub   0.000384    0.000704   1.522330
 mrms   0.000512    0.000461   1.544448 $$$
 mcpy   0.000289    0.000452   1.558333
 vcpy   0.000289    0.002705   1.561805

2 VMs, 10000000 Mbit/sec, par cost = 1.561805, speed-up = 1.12, eff = 0.56
```

For a task parallel execution variable and loop analyses are necessary. CAT results are presented below. First in the Hopfield listings, 2 Constant matrices are identified, then for each line all matrix variables are analysed. The matrices are S0, S1, HW, ST0, ST and SD. Each column beneath the matrices have symbols indicating whether the matrix is written, read, is on forward or backward data stream. Symbol 0 indicates no operation. In two digit symbols, the first digit 1 means forward flow, 2 backward flow, and 3 flow in both direction. The second digit 1 means Read, 2 Read and Write, and finally 3 means Write only. The pipe_cost column gives an estimate of communications cost in the case of splitting the representation in that line, which would result in a data flow breaking and a number of data transfers between the broken parts. The final column contains the list of backward flow data names in those specific lines. For example between the lines 27 to 33, the matrix variable ST0 is in a backward flow path, and prevents a pipeline type of task parallelism. Because of this APM efforts to divide the representation into 2 equally balanced parts cannot succeed, and APM operation results in "NO CUTTING POINT FOUND".

```
Constant S0
Constant HW
                  S0 S1 HW ST0 ST SD pipe_cost backward_flow
 24        vcpy   0  0  0 13  0  0      0.010
 25    for_JL_4   0  0  0 10  0  0      0.125
 26        mmul   0  0  0 31 13  0      0.136 ST0
 27        mtan   0  0  0 30 12  0      0.627 ST0
 28        msub   0  0  0 31 11 13      0.637 ST0
 29       mrms   0  0  0 20 10 11     1.003 STO
 30         if   0  0  0 20 10  0     0.501 STO
 31      break   0  0  0 20 10  0     0.501 STO
 32    }_endif   0  0  0 20 10  0     0.501 STO
 33       mcpy   0  0  0 33 11  0     0.627 STO
 34   }_endfor   0  0  0  0 10  0     0.501
 35      matsh   0  0  0  0 10  0     0.501
 36     printf   0  0  0  0 10  0     0.501
 37       vcpy   0 13  0  0 11  0     0.627

2 vms: total computation=       1.747
comp=        0.874 max=      0.961 min=       0.786
cutting point is searched
comp=        0.874 max=      1.048 min=       0.699
cutting point is searched
comp=        0.874 max=      1.136 min=       0.612
cutting point is searched
comp=        0.874 max=      1.223 min=       0.524
cutting point is searched
comp=        0.874 max=      1.311 min=       0.437
cutting point is searched
comp=        0.874 max=      1.398 min=       0.349
cutting point is searched
comp=        0.874 max=      1.485 min=       0.262
cutting point is searched
comp=        0.874 max=      1.573 min=       0.175
cutting point is searched
CUTTING POINT NOT FOUND
```

D.2. The Self-Organising Map

Similar to the Hopfield net explained above, here are the CAT/APM results for the SOM.

```
   0:    S0[  12 ][  64 ] ->  768
   1:    S1[  12 ][  64 ] ->  768
   2:    SE[   1 ][  12 ] ->   12
   3:    SW[  12 ][  64 ] ->  768
   4:    LW[  12 ][  12 ] ->  144
   5:    LD[  12 ][  12 ] ->  144
   6:    ST[   1 ][  64 ] ->   64
   7:    SC[   1 ][  12 ] ->   12
TOTAL MATRIX MEMORY=   2680    21440 Bytes

line func    repeat size unit   lcost       comp      total
 29: mran        1   768  7.8  0.005952    0.006     0.006
 30: mdis        1   144  7.8  0.001116    0.001     0.007
 31: mlat        1   144  9.0  0.001296    0.001     0.008
 32: mscm        1   144  6.2  0.000900    0.000     0.009
 36: vsub     4320    64  6.0  0.000384    1.659     1.668
 37: mrms     4320    64  8.0  0.000512    2.212     3.880
 38: mset     4320     1  4.5  0.000005    0.020     3.900
 40: mmin      360    12  5.0  0.000060    0.022     3.921
 41: mset      360     1  4.5  0.000005    0.002     3.923
 43: sval     4320     1  4.5  0.000005    0.020     3.942
 44: vsub     4320    64  6.0  0.000384    1.659     5.601
 45: mscm     4320    64  6.2  0.000400    1.728     7.329
 46: vadd     4320    64  6.0  0.000384    1.659     8.988
 49: mavg       30    12  6.0  0.000072    0.002     8.990
 51: mlat       30   144  9.0  0.001296    0.039     9.029
 52: mscm       30   144  6.2  0.000900    0.027     9.056
 64: vsub      144    64  6.0  0.000384    0.055     9.111
 65: mrms      144    64  8.0  0.000512    0.074     9.185
 66: mset      144     1  4.5  0.000005    0.000     9.186
 68: mmin       12    12  5.0  0.000060    0.000     9.186
 71: vcpy       12   768  4.5  0.003471    0.042     9.228
TOTAL COMPUTATIONAL COST=      9.228 seconds

func    sequential  parallel    total
 mran   0.005952   0.005434   0.005434 $$$
 mdis   0.001116   0.001019   0.006452 $$$
 mlat   0.001296   0.001339   0.007748
 mscm   0.000900   0.001141   0.008648
 vsub   0.000384   0.005210   1.667528
 mrms   0.000512   0.000461   3.658184 $$$
 mset   0.000005   0.000041   3.677711
 mmin   0.000060   0.000068   3.699311
 mset   0.000005   0.000041   3.700938
 sval   0.000005   0.000463   3.720464
 vsub   0.000384   0.005210   5.379344
 mscm   0.000400   0.000507   7.107344
 vadd   0.000384   0.005120   8.766224
 mavg   0.000072   0.000074   8.768384
 mlat   0.001296   0.001339   8.807264
 mscm   0.000900   0.001141   8.834264
 vsub   0.000384   0.005210   8.889560
 mrms   0.000512   0.000461   8.955916 $$$
 mset   0.000005   0.000041   8.956566
 mmin   0.000060   0.000068   8.957286
 vcpy   0.003471   0.005422   8.998943
   2 10000000    8.998943 ***      1.03         0.51
```

```
Constant S0
                  S0 S1 SE SW LW LD ST SC pipe_cost backward_flow
 29        mran    0  0  0 13  0  0  0  0     0.125
 30        mdis    0  0  0 10  0 13  0  0     0.149
 31        mlat    0  0  0 10 13 11  0  0     0.172
 32        mscm    0  0  0 10 12 10  0  0     0.172
 33   for_JL_30    0  0  0 10 10 10  0  0     0.172
 34   for_NP_12    0  0  0 10 10 10  0  0     0.172
 35   for_OU_12    0  0  0 10 10 10  0  0     0.172
 36        vsub    0  0  0 31 10 10 13  0     0.183 SW
 37        mrms    0  0  0 30 10 10 11  0    45.294 SW
 38        mset    0  0  0 30 10 10  0 13     0.174 SW
 39    }_endfor    0  0  0 30 10 10  0 10     8.633 SW
 40        mmin    0  0  0 30 10 10  0 11     8.633 SW
 41        mset    0  0 13 30 10 10  0  0     0.174 SW
 42   for_OU_12    0  0 10 30 10 10  0  0     0.877 SW
 43        sval    0  0 10 30 31 10  0  0     0.877 SW LW
 44        vsub    0  0 10 31 20 10 13  0    45.975 SW LW
 45        mscm    0  0 10 30 20 10 12  0    45.975 SW LW
 46        vadd    0  0 10 32 20 10 11  0    45.975 SW LW
 47    }_endfor    0  0 10 10 20 10  0  0     0.854 LW
 48    }_endfor    0  0 10 10 20 10  0  0     0.854 LW
 49        mavg    0  0 11 10 20 10  0  0     0.854 LW
 50      printf    0  0  0 10 20 10  0  0     0.149 LW
 51        mlat    0  0  0 10 33 11  0  0     0.172 LW
 52        mscm    0  0  0 10 32  0  0  0     0.830 LW
 53        dist    0  0  0 10  0  0  0  0     0.125
 54          if    0  0  0 10  0  0  0  0     0.125
 55        dist    0  0  0 10  0  0  0  0     0.125
 56     }_endif    0  0  0 10  0  0  0  0     0.125
 57        gain    0  0  0 10  0  0  0  0     0.125
 58          if    0  0  0 10  0  0  0  0     0.125
 59        gain    0  0  0 10  0  0  0  0     0.125
 60     }_endif    0  0  0 10  0  0  0  0     0.125
 61    }_endfor    0  0  0 10  0  0  0  0     0.125
 62   for_NP_12    0  0  0 10  0  0  0  0     0.125
 63   for_OU_12    0  0  0 10  0  0  0  0     0.125
 64        vsub    0  0  0 11  0  0 13  0    45.247
 65        mrms    0  0  0 10  0  0 11  0     1.629
 66        mset    0  0  0 10  0  0  0 13     8.586
 67    }_endfor    0  0  0 10  0  0  0 10     0.407
 68        mmin    0  0  0 10  0  0  0 11     0.407
 69      printf    0  0  0 10  0  0  0  0     0.125
 70       matsh    0  0  0 10  0  0  0  0     0.125
 71        vcpy    0 13  0 11  0  0  0  0     0.251

2 vms: total computation=       9.228
comp=        4.614 max=      5.075 min=       4.153
cutting point is searched
comp=        4.614 max=      5.537 min=       3.691
cutting point is searched
comp=        4.614 max=      5.998 min=       3.230
cutting point is searched
comp=        4.614 max=      6.460 min=       2.768
cutting point is searched
comp=        4.614 max=      6.921 min=       2.307
cutting point is searched
comp=        4.614 max=      7.382 min=       1.846
cutting point is searched
comp=        4.614 max=      7.844 min=       1.384
cutting point is searched
comp=        4.614 max=      8.305 min=       0.923
cutting point is searched
CUTTING POINT NOT FOUND
```

D.3. The Backpropagation Model

Below is the CAT/APM results for the Backpropagation model. A thorough explanation is given in section D.1.

```
   0:   SI[  12 ][  64 ] ->  768
   1:   Si[   1 ][  64 ] ->   64
   2:   ST[  12 ][  64 ] ->  768
   3:   SR[  12 ][  64 ] ->  768
   4:   Sr[   1 ][  64 ] ->   64
   5:   Sh[   1 ][  12 ] ->   12
   6:   PE[  12 ][  64 ] ->  768
   7:   ER[   1 ][  64 ] ->   64
   8:   W1[  64 ][  12 ] ->  768
   9:   M1[  64 ][  12 ] ->  768
  10:   W2[  12 ][  64 ] ->  768
  11:   M2[  12 ][  64 ] ->  768
  12:   A1[   1 ][  12 ] ->   12
  13:   D1[   1 ][  12 ] ->   12
  14:   E2[   1 ][  64 ] ->   64
  15:   A2[   1 ][  64 ] ->   64
  16:   D2[   1 ][  64 ] ->   64
TOTAL MATRIX MEMORY=   6564    52512 Bytes
```

```
line func    repeat size unit   lcost        comp       total
 41: mran       1    768  7.8  0.005952     0.006      0.006
 42: mran       1    768  7.8  0.005952     0.006      0.012
 45: vcpy     276     64  4.5  0.000289     0.080      0.092
 46: mmul     276    768  8.0  0.006144     1.696      1.787
 47: mtan     276     12 36.0  0.000432     0.119      1.907
 48: mmul     276    768  8.0  0.006144     1.696      3.603
 49: mtan     276     64 36.0  0.002307     0.637      4.239
 50: vsub     276     64  6.0  0.000384     0.106      4.345
 51: vcpy     276     64  4.5  0.000289     0.080      4.425
 52: dtan     276     64 12.1  0.000772     0.213      4.638
 53: memu     276     64  6.2  0.000400     0.110      4.749
 54: mscm     276     64  6.2  0.000400     0.110      4.859
 55: mscm     276    768  6.2  0.004800     1.325      6.184
 57: sval    3312      1  4.5  0.000005     0.015      6.199
 58: mscm    3312     64  6.2  0.000400     1.325      7.524
 59: vadd    3312     64  6.0  0.000384     1.272      8.795
 60: vadd    3312     64  6.0  0.000384     1.272     10.067
 61: vcpy    3312     64  4.5  0.000289     0.958     11.025
 63: mtrm     276    768  8.0  0.006144     1.696     12.721
 64: dtan     276     12 12.1  0.000145     0.040     12.761
 65: memu     276     12  6.2  0.000075     0.021     12.782
 66: mscm     276    768  6.2  0.004800     1.325     14.107
 68: sval   17664      1  4.5  0.000005     0.080     14.186
 69: mscm   17664     12  6.2  0.000075     1.325     15.511
 70: vadd   17664     12  6.0  0.000072     1.272     16.783
 71: vadd   17664     12  6.0  0.000072     1.272     18.055
 72: vcpy   17664     12  4.5  0.000054     0.958     19.013
 74: mabs     276    768  5.0  0.003840     1.060     20.073
 76: mabs      23    768  5.0  0.003840     0.088     20.161
 77: mmax      23    768  5.0  0.003840     0.088     20.249
 78: mavg      23    768  6.0  0.004608     0.106     20.355
 85: vcpy      12     64  4.5  0.000289     0.003     20.359
 86: mmul      12    768  8.0  0.006144     0.074     20.433
 87: mtan      12     12 36.0  0.000432     0.005     20.438
 88: mmul      12    768  8.0  0.006144     0.074     20.511
 89: mtan      12     64 36.0  0.002307     0.028     20.539
TOTAL COMPUTATIONAL COST=    20.539 seconds
```

```
func     sequential  parallel    total
 mran    0.005952   0.005434   0.005434 $$$
 mran    0.005952   0.005434   0.010867 $$$
 vcpy    0.000289   0.002705   0.090708
 mmul    0.006144   0.005754   1.678702 $$$
 mtan    0.000432   0.000274   1.754282 $$$
 mmul    0.006144   0.005670   3.319312 $$$
 mtan    0.002307   0.001460   3.722405 $$$
 vsub    0.000384   0.002957   3.828389
 vcpy    0.000289   0.002705   3.908230
 dtan    0.000772   0.000693   4.099620 $$$
 memu    0.000400   0.000712   4.210020
 mscm    0.000400   0.000507   4.320420
 mscm    0.004800   0.006086   5.645220
 sval    0.000005   0.000041   5.660190
 mscm    0.000400   0.000507   6.984990
 vadd    0.000384   0.002957   8.256798
 vadd    0.000384   0.005120   9.528606
 vcpy    0.000289   0.002705  10.486701
 mtrm    0.006144   0.005754  12.074695 $$$
 dtan    0.000145   0.000130  12.110580 $$$
 memu    0.000075   0.000133  12.131280
 mscm    0.004800   0.006086  13.456080
 sval    0.000005   0.000207  13.535922
 mscm    0.000075   0.000095  14.860722
 vadd    0.000072   0.002551  16.132530
 vadd    0.000072   0.004954  17.404338
 vcpy    0.000054   0.002504  18.362433
 mabs    0.003840   0.005606  19.422273
 mabs    0.003840   0.005606  19.510593
 mmax    0.003840   0.004378  19.598913
 mavg    0.004608   0.004762  19.704897
 vcpy    0.000289   0.002705  19.708368
 mmul    0.006144   0.005754  19.777411 $$$
 mtan    0.000432   0.000274  19.780698 $$$
 mmul    0.006144   0.005670  19.848742 $$$
 mtan    0.002307   0.001460  19.866268 $$$

   2 10000000  19.866268 ***     1.03         0.52

Constant SI
Constant ST
Constant SR
Constant ER

                SI  Si  ST  SR  Sr  Sh  PE  ER  W1  M1  W2  M2  A1  D1  E2  A2  D2 pipe_cost backward_flow
 41        mran   0   0   0   0   0   0   0   0  13   0   0   0   0   0   0   0   0     0.125
 42        mran   0   0   0   0   0   0   0   0  10   0  13   0   0   0   0   0   0     0.251
 43   for_JL_23   0   0   0   0   0   0   0   0  10   0  10   0   0   0   0   0   0     0.251
 44   for_NP_12   0   0   0   0   0   0   0   0  10   0  10   0   0   0   0   0   0     0.251
 45        vcpy   0  13   0   0   0   0   0   0  10   0  10   0   0   0   0   0   0     0.261
 46        mmul   0  11   0   0   0   0   0   0  31   0  10   0  13   0   0   0   0     3.135 W1
 47        mtan   0  10   0   0   0  13   0   0  30   0  10   0  11   0   0   0   0     3.676 W1
 48        mmul   0  10   0   0   0  11   0   0  30   0  31   0   0   0   0  13   0     3.684 W1 W2
 49        mtan   0  10   0   0  13  10   0   0  30   0  30   0   0   0   0  11   0     6.567 W1 W2
 50        vsub   0  10   0   0  11  10   0   0  30   0  30   0   0   0  13   0   0     6.567 W1 W2
 51        vcpy   0  10   0   0  10  10  13   0  30   0  30   0   0   0  11   0   0     9.565 W1 W2
 52        dtan   0  10   0   0  11  10  10   0  30   0  30   0   0   0  10  13   0    46.915 W1 W2
 53        memu   0  10   0   0   0  10  10   0  30   0  30   0   0   0  11  11  13    44.043 W1 W2
 54        mscm   0  10   0   0   0  10  10   0  30   0  30   0   0   0   0   0  12    41.150 W1 W2
 55        mscm   0  10   0   0   0  10  10   0  30   0  30  32   0   0   0   0  10    41.275 W1 W2 M2
 56  for_HID_12   0  10   0   0   0  10  10   0  30   0  30  30   0   0   0   0  10    41.275 W1 W2 M2
 57        sval   0  10   0   0   0  11  10   0  30   0  30  30   0   0   0   0  10    41.275 W1 W2 M2
 58        mscm   0  10   0   0   0  10  10   0  30   0  30  30   0   0   0  13  11    44.158 W1 W2 M2
 59        vadd   0  10   0   0   0  10  10   0  30   0  30  31   0   0   0  12  10    75.868 W1 W2 M2
```

```
60       vadd  0 10  0  0  0 10 10  0 30  0 32 20  0  0  0 11 10    75.743 W1 W2 M2
61       vcpy  0 10  0  0  0 10 10  0 30  0 10 33  0  0  0 11 10    75.868 W1 M2
62   }_endfor  0 10  0  0  0 10 10  0 30  0 10  0  0  0  0  0 10    41.150 W1
63       mtrm  0 10  0  0  0 10 10  0 30  0 11  0  0 13  0  0 11    41.152 W1
64       dtan  0 10  0  0  0 11 10  0 30  0 10  0 13 10  0  0  0    39.348 W1
65       memu  0 10  0  0  0  0 10  0 30  0 10  0 11 12  0  0  0    38.808 W1
66       mscm  0 10  0  0  0  0 10  0 30 32 10  0  0 10  0  0  0    38.392 W1 M1
67  for_IN_64  0 10  0  0  0  0 10  0 30 30 10  0  0 10  0  0  0    38.392 W1 M1
68       sval  0 11  0  0  0  0 10  0 30 30 10  0  0 10  0  0  0    38.392 W1 M1
69       mscm  0  0  0  0  0  0 10  0 30 30 10  0 13 11  0  0  0    36.050 W1 M1
70       vadd  0  0  0  0  0  0 10  0 30 31 10  0 12  0  0  0  0    69.562 W1 M1
71       vadd  0  0  0  0  0  0 10  0 32 20 10  0 11  0  0  0  0    69.437 W1 M1
72       vcpy  0  0  0  0  0  0 10  0 10 33 10  0 11  0  0  0  0    69.562 M1
73   }_endfor  0  0  0  0  0  0 10  0 10 30 10  0  0  0  0  0  0  2248.807 M1
74       mabs  0  0  0  0  0  0 10  0 10 32 10  0  0  0  0  0  0  2248.807 M1
75   }_endfor  0  0  0  0  0  0 10  0 10  0 10  0  0  0  0  0  0    34.844
76       mabs  0  0  0  0  0  0 12  0 10  0 10  0  0  0  0  0  0    34.844
77       mmax  0  0  0  0  0  0 11  0 10  0 10  0  0  0  0  0  0    34.844
78       mavg  0  0  0  0  0  0 11  0 10  0 10  0  0  0  0  0  0    34.844
79     printf  0  0  0  0  0  0  0  0 10  0 10  0  0  0  0  0  0     0.251
80         if  0  0  0  0  0  0  0  0 10  0 10  0  0  0  0  0  0     0.251
81      break  0  0  0  0  0  0  0  0 10  0 10  0  0  0  0  0  0     0.251
82    }_endif  0  0  0  0  0  0  0  0 10  0 10  0  0  0  0  0  0     0.251
83   }_endfor  0  0  0  0  0  0  0  0 10  0 10  0  0  0  0  0  0     0.251
84  for_NP_12  0  0  0  0  0  0  0  0 10  0 10  0  0  0  0  0  0     0.251
85       vcpy  0 13  0  0  0  0  0  0 10  0 10  0  0  0  0  0  0     3.133
86       mmul  0 11  0  0  0  0  0  0 11  0 10  0 13  0  0  0  0    34.969
87       mtan  0  0  0  0  0 13  0  0  0  0 10  0 11  0  0  0  0     0.689
88       mmul  0  0  0  0  0 11  0  0  0  0 11  0  0  0  0 13  0    34.742
89       mtan  0  0  0  0 13  0  0  0  0  0  0  0  0  0  0 11  0     3.008

2 vms: total computation=       9.228
comp=        4.614 max=       5.075 min=        4.153
cutting point is searched
comp=        4.614 max=       5.537 min=        3.691
cutting point is searched
comp=        4.614 max=       5.998 min=        3.230
cutting point is searched
comp=        4.614 max=       6.460 min=        2.768
cutting point is searched
comp=        4.614 max=       6.921 min=        2.307
cutting point is searched
comp=        4.614 max=       7.382 min=        1.846
cutting point is searched
comp=        4.614 max=       7.844 min=        1.384
cutting point is searched
comp=        4.614 max=       8.305 min=        0.923
cutting point is searched
CUTTING POINT NOT FOUND
```

D.4. The SOM/Backpropagation Model

The same technique is applied to the cooperating SOM and Backpropagation networks.

```
The SOM/Backpropagation Network

   0:   S0[  12 ][  64 ] ->  768
   1:   SM[   1 ][  64 ] ->   64
   2:   S1[  12 ][  64 ] ->  768
   3:   SE[   1 ][  12 ] ->   12
   4:   WI[   1 ][  12 ] ->   12
   5:   SW[  12 ][  64 ] ->  768
   6:   DW[  12 ][  64 ] ->  768
   7:   LW[  12 ][  12 ] ->  144
   8:   LD[  12 ][  12 ] ->  144
   9:   SC[   1 ][  12 ] ->   12
  10:   Si[   1 ][  64 ] ->   64
  11:   ST[  12 ][  64 ] ->  768
  12:   SR[  12 ][  64 ] ->  768
  13:   Sr[   1 ][  64 ] ->   64
  14:   Sh[   1 ][  24 ] ->   24
  15:   PE[  12 ][  64 ] ->  768
  16:   ER[   1 ][  64 ] ->   64
  17:   W1[  64 ][  24 ] -> 1536
  18:   M1[  64 ][  24 ] -> 1536
  19:   W2[  24 ][  64 ] -> 1536
  20:   M2[  24 ][  64 ] -> 1536
  21:   A1[   1 ][  24 ] ->   24
  22:   D1[   1 ][  24 ] ->   24
  23:   E2[   1 ][  64 ] ->   64
  24:   A2[   1 ][  64 ] ->   64
  25:   D2[   1 ][  64 ] ->   64
TOTAL MATRIX MEMORY=  12364    98912 Bytes
```

```
line func   repeat size unit   lcost       comp      total
 64: mran      1   1536  7.8  0.011904    0.012      0.012
 65: mran      1   1536  7.8  0.011904    0.012      0.024
 68: mran      1    768  7.8  0.005952    0.006      0.030
 69: mdis      1    144  7.8  0.001116    0.001      0.031
 70: mlat      1    144  9.0  0.001296    0.001      0.032
 71: mscm      1    144  6.2  0.000900    0.000      0.033
 75: vsub   4320     64  6.0  0.000384    1.659      1.692
 76: mrms   4320     64  8.0  0.000512    2.212      3.904
 77: mset   4320      1  4.5  0.000005    0.020      3.923
 79: mmin    360     12  5.0  0.000060    0.022      3.945
 80: mset    360      1  4.5  0.000005    0.002      3.947
 81: mset    360      1  4.5  0.000005    0.002      3.948
 83: sval   4320      1  4.5  0.000005    0.020      3.968
 84: vsub   4320     64  6.0  0.000384    1.659      5.627
 85: mscm   4320     64  6.2  0.000400    1.728      7.355
 86: vadd   4320     64  6.0  0.000384    1.659      9.013
 89: mavg     30     12  6.0  0.000072    0.002      9.016
 91: mlat     30    144  9.0  0.001296    0.039      9.054
 92: mscm     30    144  6.2  0.000900    0.027      9.081
102: sval    360      1  4.5  0.000005    0.002      9.083
104: vcpy    360     64  4.5  0.000289    0.104      9.187
105: mmul    360   1536  8.0  0.012288    4.424     13.611
106: mtan    360     24 36.0  0.000865    0.311     13.922
107: mmul    360   1536  8.0  0.012288    4.424     18.346
108: mtan    360     64 36.0  0.002307    0.830     19.176
109: vsub    360     64  6.0  0.000384    0.138     19.315
```

```
110: vcpy     360     64  4.5  0.000289      0.104      19.419
111: dtan     360     64 12.1  0.000772      0.278      19.697
112: memu     360     64  6.2  0.000400      0.144      19.841
113: mscm     360     64  6.2  0.000400      0.144      19.985
114: mscm     360   1536  6.2  0.009600      3.456      23.441
116: sval    8640      1  4.5  0.000005      0.039      23.480
117: mscm    8640     64  6.2  0.000400      3.456      26.936
118: vadd    8640     64  6.0  0.000384      3.318      30.254
119: vadd    8640     64  6.0  0.000384      3.318      33.571
120: vcpy    8640     64  4.5  0.000289      2.499      36.071
122: mtrm     360   1536  8.0  0.012288      4.424      40.494
123: dtan     360     24 12.1  0.000290      0.104      40.599
124: memu     360     24  6.2  0.000150      0.054      40.653
125: mscm     360   1536  6.2  0.009600      3.456      44.109
127: sval   23040      1  4.5  0.000005      0.104      44.213
128: mscm   23040     24  6.2  0.000150      3.456      47.669
129: vadd   23040     24  6.0  0.000144      3.318      50.987
130: vadd   23040     24  6.0  0.000144      3.318      54.304
131: vcpy   23040     24  4.5  0.000108      2.499      56.804
133: mabs     360   1536  5.0  0.007680      2.765      59.569
135: mabs      30    768  5.0  0.003840      0.115      59.684
136: mmax      30    768  5.0  0.003840      0.115      59.799
137: mavg      30    768  6.0  0.004608      0.138      59.937
TOTAL COMPUTATIONAL COST=    59.937 seconds

func     sequential  parallel    total
 mran    0.011904    0.010867    0.010867 $$$
 mran    0.011904    0.010867    0.021734 $$$
 mran    0.005952    0.005434    0.027168 $$$
 mdis    0.001116    0.001019    0.028187 $$$
 mlat    0.001296    0.001339    0.029483
 mscm    0.000900    0.001141    0.030383
 vsub    0.000384    0.005210    1.689263
 mrms    0.000512    0.000461    3.679919 $$$
 mset    0.000005    0.000041    3.699445
 mmin    0.000060    0.000068    3.721045
 mset    0.000005    0.000041    3.722672
 mset    0.000005    0.000041    3.724300
 sval    0.000005    0.000463    3.743826
 vsub    0.000384    0.005210    5.402706
 mscm    0.000400    0.000507    7.130706
 vadd    0.000384    0.005120    8.789586
 mavg    0.000072    0.000074    8.791746
 mlat    0.001296    0.001339    8.830626
 mscm    0.000900    0.001141    8.857626
 sval    0.000005    0.000041    8.859253
 vcpy    0.000289    0.002705    8.963394
 mmul    0.012288    0.011302   13.032258 $$$
 mtan    0.000865    0.000548   13.229423 $$$
 mmul    0.012288    0.011238   17.275247 $$$
 mtan    0.002307    0.001460   17.801020 $$$
 vsub    0.000384    0.002957   17.939260
 vcpy    0.000289    0.002705   18.043400
 dtan    0.000772    0.000693   18.293039 $$$
 memu    0.000400    0.000712   18.437039
 mscm    0.000400    0.000507   18.581039
 mscm    0.009600    0.012173   22.037039
 sval    0.000005    0.000079   22.076092
 mscm    0.000400    0.000507   25.532092
 vadd    0.000384    0.005414   28.849852
 vadd    0.000384    0.010035   32.167612
 vcpy    0.000289    0.005162   34.666991
 mtrm    0.012288    0.011302   38.735855 $$$
 dtan    0.000290    0.000260   38.829469 $$$
 memu    0.000150    0.000267   38.883469
```

```
mscm    0.009600    0.012173   42.339469
sval    0.000005    0.000207   42.443610
mscm    0.000150    0.000190   45.899610
vadd    0.000144    0.005102   49.217370
vadd    0.000144    0.009907   52.535130
vcpy    0.000108    0.005008   55.034509
mabs    0.007680    0.011213   57.799309
mabs    0.003840    0.005606   57.914509
mmax    0.003840    0.004378   58.029709
mavg    0.004608    0.004762   58.167949

  2 10000000   58.167949 ***      1.03          0.52
```

Constants: S0 S1 DW ST SR ER

		S0	SM	S1	SE	WI	SW	DW	LW	LD	SC	Si	ST	SR	Sr	Sh	PE	ER	W1	M1	W2	M2	A1	D1	E2	A2	D2	pipe_cost	backward_flow
64	mran	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0	0	0.251	
65	mran	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	13	0	0	0	0	0	0	0.501	
66	ran	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.501	
67	matld	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.501	
68	mran	0	0	0	0	0	13	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.627	
69	mdis	0	0	0	0	0	10	0	0	13	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.650	
70	mlat	0	0	0	0	0	10	0	13	11	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.674	
71	mscm	0	0	0	0	0	10	0	12	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.674	
72	for_JL_30	0	0	0	0	0	10	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.674	
73	for_NP_12	0	0	0	0	0	10	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.674	
74	for_OU_12	0	0	0	0	0	10	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.674	
75	vsub	0	13	0	0	0	31	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.684	SW
76	mrms	0	11	0	0	0	30	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	45.795	SW
77	mset	0	0	0	0	0	30	0	10	10	13	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.676	SW
78	}_endfor	0	0	0	0	0	30	0	10	10	10	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	9.134	SW
79	mmin	0	0	0	0	0	30	0	10	10	11	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	9.134	SW
80	mset	0	0	0	0	13	30	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.676	SW
81	mset	0	0	0	13	10	30	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.381	SW
82	for_OU_12	0	0	0	10	10	30	0	10	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	2.084	SW
83	sval	0	0	0	10	10	30	0	31	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	2.084	SW LW
84	vsub	0	13	0	10	10	31	0	20	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	47.182	SW LW
85	mscm	0	12	0	10	10	30	0	20	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	47.182	SW LW
86	vadd	0	11	0	10	10	32	0	20	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	47.182	SW LW
87	}_endfor	0	0	0	10	10	10	0	20	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	2.060	LW
88	}_endfor	0	0	0	10	10	10	0	20	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	2.060	LW
89	mavg	0	0	0	11	10	10	0	20	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	2.060	LW
90	printf	0	0	0	0	10	10	0	20	10	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.355	LW
91	mlat	0	0	0	0	10	10	0	33	11	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.379	LW
92	mscm	0	0	0	0	10	10	0	32	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	2.037	LW
93	dist	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
94	if	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
95	dist	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
96	}_endif	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
97	gain	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
98	if	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
99	gain	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
100	}_endif	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
101	for_NP_12	0	0	0	0	10	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
102	sval	0	0	0	0	11	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	1.332	
103	col	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.627	
104	vcpy	0	0	0	0	0	11	0	0	0	0	13	0	0	0	0	0	0	10	0	10	0	0	0	0	0	0	0.637	
105	mmul	0	0	0	0	0	0	0	0	0	0	11	0	0	0	0	0	0	31	0	10	0	13	0	0	0	0	4.265	W1
106	mtan	0	0	0	0	0	0	0	0	0	0	10	0	0	0	13	0	0	30	0	10	0	11	0	0	0	0	5.675	W1
107	mmul	0	0	0	0	0	0	0	0	0	0	10	0	0	0	11	0	0	30	0	31	0	0	0	0	13	0	5.682	W1 W2
108	mtan	0	0	0	0	0	0	0	0	0	0	10	0	0	13	10	0	0	30	0	30	0	0	0	0	11	0	9.442	W1 W2
109	vsub	0	0	0	0	0	0	0	0	0	0	10	0	0	11	10	0	0	30	0	30	0	0	0	13	0	0	9.442	W1 W2
110	vcpy	0	0	0	0	0	0	0	0	0	0	10	0	0	10	10	13	0	30	0	30	0	0	0	11	0	0	13.317	W1 W2
111	dtan	0	0	0	0	0	0	0	0	0	0	10	0	0	11	10	10	0	30	0	30	0	0	0	10	13	0	62.073	W1 W2
112	memu	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	30	0	0	0	11	11	13	58.324	W1 W2
113	mscm	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	30	0	0	0	0	0	12	54.553	W1 W2
114	mscm	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	30	32	0	0	0	0	10	54.804	W1 W2 M2
115	for_HID_24	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	30	30	0	0	0	0	10	54.804	W1 W2 M2
116	sval	0	0	0	0	0	0	0	0	0	0	10	0	0	0	11	10	0	30	0	30	30	0	0	0	0	10	54.804	W1 W2 M2
117	mscm	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	30	30	0	0	0	13	11	58.564	W1 W2 M2
118	vadd	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	30	31	0	0	0	12	10	145.047	W1 W2 M2
119	vadd	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	32	20	0	0	0	11	10	144.796	W1 W2 M2
120	vcpy	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	10	33	0	0	0	11	10	145.047	W1 M2
121	}_endfor	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	10	0	0	0	0	0	10	54.553	W1
122	mtrm	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10	10	0	30	0	11	0	0	13	0	0	11	54.557	W1
123	dtan	0	0	0	0	0	0	0	0	0	0	10	0	0	0	11	10	0	30	0	0	0	13	10	0	0	0	53.362	W1
124	memu	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	10	0	30	0	0	0	11	12	0	0	0	51.952	W1
125	mscm	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	10	0	30	32	0	0	0	10	0	0	0	50.793	W1 M1
126	for_BIN_64	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	10	0	30	30	0	0	0	10	0	0	0	50.793	W1 M1
127	sval	0	0	0	0	0	0	0	0	0	0	11	0	0	0	0	10	0	30	30	0	0	0	10	0	0	0	50.793	W1 M1
128	mscm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	30	30	0	0	13	11	0	0	0	48.443	W1 M1
129	vadd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	30	31	0	0	12	0	0	0	0	135.866	W1 M1
130	vadd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	32	20	0	0	11	0	0	0	0	135.615	W1 M1
131	vcpy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	33	0	0	11	0	0	0	0	135.615	M1
132	}_endfor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	30	0	0	0	0	0	0	0	5820.678	M1
133	mabs	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	32	0	0	0	0	0	0	0	5820.678	M1
134	}_endfor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	45.122	
135	mabs	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	45.122	
136	mmax	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0	0	0	0	0	0	0	0	0	45.122	
137	mavg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0	0	0	0	0	0	0	0	0	45.122	

```
2 vms: total computation=     59.937
comp=      29.969 max=     32.965 min=      26.972
cutting point is searched
comp=      29.969 max=     35.962 min=      23.975
cutting point is searched
comp=      29.969 max=     38.959 min=      20.978
cutting point is searched
comp=      29.969 max=     41.956 min=      17.981
cutting point is searched
comp=      29.969 max=     44.953 min=      14.984
cutting point is searched
comp=      29.969 max=     47.950 min=      11.987
cutting point is searched
comp=      29.969 max=     50.947 min=       8.991
Cut line:102 comcosts=      0.627 achieved=       9.083
```

The composite listing of the SOM/Backpropagation program is split into two parts at line 102. Cutting for this network configuration results in 9.803 sec. computational load on the first VM and the rest (59.37 - 9.08) sec. on the second VM. The two *MATLIB* listings can be pipelined through two parallel machines. A server program running on the third processor can be used to carry out File I/O and data transfer between the two VMs. The programs are as follows;

```
#include "uv.h"          /* The server program */
int    NP = 12;
int    IN = 64;
int    OU = 12;
int    JL = 30;
int      BPIN = 64;
int      HID = 12;
int      OUT = 64;
main()
{
       int    i;
       mt_type *SW, *W1, *W2;
       int    *vm;

       matdef(&SW,  "SW",  OU,    IN);
         matdef(&W1,      "W1",   BPIN,    HID);
         matdef(&W2,      "W2",   HID,     OUT);

         mran( SW, 0.0, 0.05);
         mran( W1, -0.05, 0.05);
         mran( W2, -0.05, 0.05);

       opensocket( &vm, 2);

       put_mat( vm[1], SW);
       put_mat( vm[2], W1);
       put_mat( vm[2], W2);

       for (i=0; i<JL; i++) {
              servis( vm, 1 );
       }
}
/*-------------------------------------------------------------*/
```

The two client listings are as follows;

```
#include "uv.h"          /* Client 1 listing */
int    NP = 12;
int    IN = 64;
int    OU = 12;
int    JL = 30;
main()
{
        double  dist = 2.0;
        double  dstp = 0.4;
        double  dend = 0.1;
        double  gain = 0.9;
        double  gstp = 0.1;
        double  gend = 0.1;
          char    *infile="r.dat";
        mt_type *S0, *S1, *WI, *SW, *DW, *LW, *LD, *SM, *SC, *SE;
        int    i, j, k, n;
        int    row, col;
        doublerms, avg;
        int    fd, cli_no = 1;

          matdef(&S0, "S0", NP,   IN);
          matdef(&S1, "S1", NP,   IN);
          matdef(&SE, "SE", 1,    NP);
          matdef(&WI, "WI", 1,    NP);
          matdef(&SW, "SW", OU,   IN);
          matdef(&DW, "DW", OU,   IN);
          matdef(&LW, "LW", OU,   OU);
          matdef(&LD, "LD", OU,   OU);
          matdef(&SM, "SM", 1,    IN);
          matdef(&SC, "SC", 1,    OU);

        consocket(&fd, cli_no);

        mdis( LD, 2);
          mlat( LW, LD, dist);
          mscm( LW, LW, gain);
          matld(S0, infile);

        get_mat( SW, fd );
        for (n=0; n<JL; n++) {
                for (j=0; j<NP; j++) {
                        for (k=0; k<OU; k++) {
                                vsub(SM, S0, SW, 0, j, k);
                                vcpy(DW, SM, k, 0);
                                mrms(&rms, SM);
                                mset(SC, rms, 0, k);
                        }
                        mmin(&rms, SC, &row, &col);
                        mset(WI, (double) col, 0, j);
                        mset(SE, rms, 0, j);
                        for (k=0; k<OU; k++) {
                                sval(&rms, LW, k, col);
                                vcpy(SM, DW, 0, k);
                                mscm(SM, SM, rms);
                                vcpy(DW, SM, k, 0);
                        }
                        madd(SW, SW, DW);
                }
                mavg(&avg, SE);
                printf ("%d: avg rms = %f0, n, avg);
                mlat( LW, LD, dist);
```

```
            mscm( LW, LW, gain);
            dist = dist - dstp;
            if ( dist < dend ) {
                  dist = dend;
            }
            gain = gain - gstp;
            if ( gain < gend ) {
                  gain = gend;
            }
            post ( fd, 2, WI );
            post ( fd, 2, SW );
        }
}     /* End of Main for the Client 1*/
```

```
#include "uv.h"          /* Client 2 listing */
int    NP = 12;
int    JL = 30;
int      IN = 64;
int      OU = 12;
int    BPIN = 64;
int    HID = 12;
int    OUT = 64;
main()
{
       double  bpgain = 0.003;
       double  momt = 0.5;
       doubletol = 0.1;
       double  rms = 0.0;
       doubleavg;
       doublemax;
       int    i, j, n, k, p;
       int    row, col;
       doubletmp;
         char     *tarfile="h.dat";

       mt_type *WI, *SW;
       mt_type      *SI, *Si, *ST, *Sr, *Sh, *PE;
       mt_type      *W1, *W2, *WT, *A1, *A2, *ER;
       mt_type *M1, *M2, *E2, *D1, *D2, *SR;
              int      fd, cli_no = 2;

         consocket(&fd, cli_no);

       matdef(&WI, "WI", 1,     NP);
         matdef(&SW, "SW", OU,   IN);

       matdef(&Si,  "Si", 1,    BPIN);
       matdef(&ST,  "ST", NP,   OUT);
       matdef(&SR,  "SR", NP,   OUT);
       matdef(&Sr,  "Sr", 1,    OUT);
       matdef(&Sh,  "Sh", 1,    HID);
       matdef(&PE,  "PE", NP,   OUT);
       matdef(&ER,  "ER", 1,    OUT);
       matdef(&W1,  "W1", BPIN, HID);
       matdef(&M1,  "M1", BPIN, HID);
       matdef(&W2,  "W2", HID,  OUT);
       matdef(&M2,  "M2", HID,  OUT);
       matdef(&WT,  "WT", OUT,  HID);
       matdef(&A1,  "A1", 1,    HID);
       matdef(&D1,  "D1", 1,    HID);
       matdef(&E2,  "E2", 1,    OUT);
       matdef(&A2,  "A2", 1,    OUT);
       matdef(&D2,  "D2", 1,    OUT);

         matld(ST, tarfile);
       get_mat ( W1, fd );
       get_mat ( W2, fd );
       for (n=0; n<JL; n++) {
              get_mat( WI, fd );
              get_mat( SW, fd );
              for (p=0; p<NP; p++) {
                     sval(&rms, WI, 0, p );
                     col = (int)rms;
                     vcpy(Si, SW, 0, col);
/*
                     matsh ("", SO, p, 0.0, 8);
                     matsh ("", Si, 0, 0.0, 8);
                     matsh ("", ST, p, 0.0, 8);
*/
```

```
                   mmul(A1, Si, W1);
                   mtan(Sh, A1);
                   mmul(A2, Sh, W2);
                   mtan(Sr, A2);
                   vsub(E2, ST, Sr, 0, p, 0);
                   vcpy(PE, E2, p, 0);
                   dtan(A2, Sr);
                   memu(D2, E2, A2);
                   mscm(D2, D2, bpgain);
                   mscm(M2, M2, momt);
                   for (i=0; i<HID; i++) {
                         sval(&tmp, Sh, 0, i);
                         mscm(A2, D2, tmp);
                         vadd(A2, A2, M2, 0, 0, i);
                         vadd(W2, W2, A2, i, i, 0);
                         vcpy(M2, A2, i, 0);
                   }
                   mtra(WT, W2);
                   mmul(D1, D2, WT);
                   dtan(A1, Sh);
                   memu(D1, D1, A1);
                   mscm(M1, M1, momt);
                   for (i=0; i<BPIN; i++) {
                         sval(&tmp, Si, 0, i);
                         mscm(A1, D1, tmp);
                         vadd(A1, A1, M1, 0, 0, i);
                         vadd(W1, W1, A1, i, i, 0);
                         vcpy(M1, A1, i, 0);
                   }
                   mabs(M1, M1);
             }
                mabs(PE, PE);
                mmax(&max, PE, &row, &col);
                mavg(&avg, PE);
                printf("%4d: max=%f avg=%f tol=%f0, n, max, avg, tol);
                if ( max < tol ) {
                        break;
                }
      }
}
/*---------------------------------------------------------------*/
```

Appendix E

APPENDIX E - Competing Backpropagation Networks

This appendix presents the NETLIB listings of the competing Backpropagation networks. Two listings show the server and the client programs using parallel features of the NETLIB.

The first program listing is the server *NETLIB* listing. As can be seen the server opens a socket for 2 clients. For input (SI) and target (ST) matrices, memory is allocated and the matrix structures are set up. Both matrices are loaded from the *infile* and *tarfile.* and transmitted to the clients. The model is generic and it is independent of the number of clients. The server code is as follows;

```
#include "uv.h"
int    NP = 12;
int      BPIN = 64;
int      OUT = 64;
char   *infile = "r.dat";
char   *tarfile= "h.dat";
main()
{
       int    i;
       mt_type *SI;
       mt_type *ST;
       int    *vm, total = 2;
       doubleerr1, err2;

       opensocket( &vm, total);

       matdef(&SI, "SI", NP,    BPIN);
       matdef(&ST, "ST", NP,    OUT);

       matld ( SI, infile );
       matld ( ST, tarfile );

       for (i=1; i<=total; i++) {
              put_mat( vm[i], SI);
              put_mat( vm[i], ST);
       }
}
/*---------------------------------------------------------*/
```

The clients are self-contained Backpropagation networks. They connect to the open socket, receive the input and target patterns, randomise weight matrices and execute their *NETLIB bplearn* and *recall* functions. Each client can call bplearn function with a different set of parameters and a network topology. The results, or the error can be monitored if wished. The clients *NETLIB* definitions are as follows;

```
#include "uv.h"
int    NP = 12;
int    IN = 64;
int    HID = 12;
int    OUT = 64;
mt_type      *SI, *ST, *W1, *W2, *SR;
int    p, fd, cli_no = 1;
main()
{
        consocket(&fd, cli_no);

      matdef(&SI,  "SI", NP,   IN);
      matdef(&ST,  "ST", NP,   OUT);
      matdef(&SR,  "SR", NP,   OUT);
      matdef(&W1,  "W1", IN,   HID);
      matdef(&W2,  "W2", HID,  OUT);

      get_mat( SI, fd );
      get_mat( ST, fd );

      mran( W1, -0.05, 0.05);
      mran( W2, -0.05, 0.05);

      bplearn ( SI, ST, W1, W2, 23, 0.003, 0.5, 0.1 );
      bprecall ( SI, SR, W1, W2 );

        for (p=0; p<NP; p++) {
             printf( "Result %d0, p);
                matsh("", SI, p, 0.0, 8);
                matsh("", SR, p, 0.0, 8);
        }
}
/*-----------------------------------------------------------*/
```