DECOMPOSING & MAPPING NEURAL SYSTEMS
ONTO
GENERAL-PURPOSE PARALLEL MACHINES

Magali E. Azema Barac

Submitted for the degree of
Doctor of Philosophy in Computer Science
University of London

Department of Computer Science
University College London
December 1993
ABSTRACT

Simulations of neural systems on sequential computers are computationally expensive. For example, a single experiment for a typical financial application, e.g. exchange rate time series analysis, requires about ten hours of CPU time on a Sun workstation. Neural systems are, however inherently parallel, and would thus benefit from parallel implementations. Therefore, this thesis investigates the problem of decomposing and mapping neural systems onto general-purpose parallel machines. It presents a Mapping System capable of decomposing neural network systems, and mapping them onto a range of general-purpose parallel machines; both MIMD and SIMD.

Firstly, taxonomies of neural network systems and parallel machines are provided, as well as descriptions of typical examples of both. Secondly, parallelism in neural systems and machines is analysed. The different types of parallelism exhibited by neural systems are identified. This allows a classification of neural systems on the basis of their parallelism and in accordance with their taxonomy. Parallel machines and the approaches to parallel programming are then analysed to identify the features of parallel machines involved in the mapping process.

From this analysis of parallelism in neural systems and machines, the characteristics required for decomposing and mapping neural systems are identified, and a Mapping System for decomposing and mapping neural systems onto general-purpose parallel machines is described. The Mapping System consists of two modules: a machine independent Abstract Decomposition module, and a Machine Dependent Decomposition module. The Abstract Decomposition (AD) module describes a specification for neural systems. The AD specifies a finite set of schemes for decomposing neural systems according to the required exploitation of neural systems' parallelism; e.g. connection, neuron, cluster. The Machine Dependent Decomposition (MDD) analyses the decomposition schemes in conjunction with the features of parallel machines; e.g. processors' features, communications schemes, and specifies the most suitable mapping scheme to implement.

To validate the Mapping System, prototype mapping software for MIMD machines has been implemented. The MIMD mapping software is able to automatically map static neural systems onto a 48-processor Parsys SN1000 Transputer machine. This mapping software was developed as part of the CEC-funded Esprit II Pygmalion Neurocomputing Project, and is incorporated in the Pygmalion Neural Network Environment.

The Machine Dependent Decomposition (MDD) module is improved by the development of an analytical framework for evaluating the speedup of neural systems' mapping schemes, based on the integration of machine-dependent features with the alternative decomposition schemes. The various mapping schemes for the classical backpropagation neural systems were hand-coded onto a 64x64-processor Distributed Array of Processors (DAP). The analytical framework is then used to evaluate the speedups of the different mapping schemes. This shows that the expected speedups agree with the results obtained when implementing the mapping schemes.

A formal specification for neural network systems which uses the Abstract Syntax Notation One (ASN.1) as the syntactic construct is then presented. The innovative use of ASN.1, previously dedicated to the specification of communication protocols by the Open System Interconnection, provides a formal basis for specifying neural systems and their parallelism.

This thesis develops a solution for decomposing and mapping neural systems onto general-purpose parallel machines. Working top down, this thesis makes three major contributions: i) the innovative use of ASN.1 as the notational support for specifying neural systems with explicit support for parallelism, ii) an analytical framework for evaluating the speedup of alternative neural system' mappings, and iii) a general-purpose Mapping System for mapping neural systems onto parallel machines.
I wish to thank my supervisor Professor Philip C. Treleaven for his help and support throughout this research. I am indebted to Dr. A. N. Refenes who read various drafts of this manuscript and provided invaluable criticism. My thanks are also due to Professor John A. Campbell for his helpful comments on this thesis.
Table of Contents

1. Introduction ............................................................................................. 9
   1.1. Neural Computing ............................................................................................... 9
       1.1.1. Neural Network Systems ......................................................................... 9
       1.1.2. Neural Computing Field .......................................................................... 13
   1.2. Parallel Computing ............................................................................................. 14
       1.2.1. Neural Computing and Parallel Computing ............................................. 14
       1.2.2. Parallel Machines ..................................................................................... 16
   1.3. Thesis Motivations & Aims ................................................................................. 19
   1.4. Thesis Outline ..................................................................................................... 21
   1.5. Thesis Contributions ........................................................................................... 23

2. Neural Network Systems & Parallel Machines ............................................................ 24
   2.1. Related Work ...................................................................................................... 24
   2.2. Neural Networks ................................................................................................. 26
       2.2.1. Survey of Neural Network Systems .......................................................... 26
       2.2.2. Taxonomy ............................................................................................... 30
   2.3. Parallel Machines ................................................................................................ 32
       2.3.1. Survey of Parallel Machines .................................................................... 32
       2.3.2. Taxonomy ............................................................................................... 35
   2.4. Summary ............................................................................................................. 36

3. Parallelism in Neural Systems & Parallel Machines ................................................... 37
   3.1. Overview ............................................................................................................. 37
   3.2. Parallelism in Neural Systems ............................................................................. 38
       3.2.1. Types of Parallelism in Neural Systems ................................................... 38
       3.2.2. Parallelism across Neural Systems ........................................................... 41
       3.2.3. Conclusion .............................................................................................. 42
   3.3. Parallelism in Machines ....................................................................................... 43
       3.3.1. Types of Parallelism ....................................................................... 43
       3.3.2. Parallelism in Languages ......................................................................... 44
       3.3.3. Approaches for Programming & Exploiting Parallel Machines .......... 47
   3.4. Mapping Neural Systems onto Parallel Machines ................................................ 49
       3.4.1. Expression of Parallelism ........................................................................ 49
       3.4.2. Exploitation of Parallelism ...................................................................... 50
   3.5. Summary ............................................................................................................. 51

4. The Mapping System .................................................................................................... 53
List of Figures

Figure 1.1 - Neural Network Model, Application and System - .......................................................... 10
Figure 1.2 - Basic Elements of a Neural Network - ........................................................................ 11
Figure 1.3 - Classification of Parallel Machine Architectures - ....................................................... 17
Figure 1.4 - System for Decomposing & Mapping Neural Systems onto Parallel Machines - .......... 21
Figure 2.1 - The SN1000 Parsys Machine - .................................................................................... 33
Figure 2.2 - The Connection Machine - .......................................................................................... 34
Figure 2.3 - The DAP Machine - ..................................................................................................... 35
Figure 3.1 - Decomposing and Mapping Neural Systems onto Parallel Machines - ...................... 37
Figure 3.2 - Types of Parallelism versus Neural Systems' Components - ....................................... 38
Figure 3.3 - Intra-neuron Parallelism - ........................................................................................... 39
Figure 3.4 - Training Parallelism - .................................................................................................. 40
Figure 3.5 - Matrix Multiplication on Transputer Grid - .................................................................. 45
Figure 3.6 - Matrix Multiplication on Processor Array - .................................................................. 46
Figure 3.7 - Expressing & Exploiting the Parallelism of Neural Systems - .................................... 52
Figure 3.8 - The System for Mapping Neural Systems - ................................................................. 55
Figure 4.1 - The System for Mapping Neural Systems - ................................................................. 55
Figure 4.2 - The nc Data Structure & Declaration of the Neuron C Data Structure - .................. 57
Figure 4.3 - Rule Body of a Standard Neuron State Update - ......................................................... 58
Figure 4.4 - Rule Class: Examples - ............................................................................................... 59
Figure 4.5 - The Initialisation of a Data Rule - ................................................................................ 59
Figure 4.6(a) - The Extension of a Data Rule - ................................................................................ 60
Figure 4.6(b) - The Extension of a Rule Applying to other Rules - ................................................ 60
Figure 4.7 - Connection Decomposition Scheme - ......................................................................... 65
Figure 4.8 - Neuron Decomposition Scheme - ................................................................................ 65
Figure 4.9 - Net Replication Scheme - ........................................................................................... 65
Figure 4.10 - Combination of Neuron Decomposition & Net Replication Schemes - ................... 66
Figure 4.11 - The MDD Module: Selection & Translation - ............................................................ 67
Figure 4.12 - Neural Network Mapping Schemes: Selection Process - .......................................... 72
Figure 5.1 - PYGMALION Neural Network Programming Environment - ...................................... 75
Figure 5.2 - Design of the Mapping Software - .............................................................................. 77
Figure 5.3 - TINY + 3L Parallel C Environment - ......................................................................... 78
Figure 5.4 - Example of Sequential Computation S - .................................................................... 80
Figure 5.5 - Example of a Parallel Computation P - ...................................................................... 80
Figure 5.6 - Computation Tree and Processing-based Decomposition - ........................................ 81
Figure 5.7 - From A Neural Network System to its Transputer-Based Machine Specification - .... 84
Figure 5.8 - An Example of Keep-init Mechanism - ..................................................................... 86
List of Tables

Table 1.1 - A Framework for Neural Network System .................................................. 12
Table 2.1 - Neural Network Systems: Common Features & Examples .......................... 31
Table 2.2 - General Purpose Parallel Machines: Common Features & Examples .......... 36
Table 3.1 - Types of Parallelism in Neural Systems ..................................................... 40
Table 3.2 - Neural Systems & Types of Parallelism ..................................................... 42
Table 4.1 - Spatial & Temporal Information for Decomposition Schemes .................... 63
Table 4.2 - Basic Mapping Schemes & Machines Features .......................................... 70
Table 5.1 - Decomposition Schemes on MIMD machines .......................................... 82
Table 6.1 - Spatial & Temporal Information for the Basic Decomposition Schemes ....... 111
Table 6.2 - The Set of Mapping Schemes to be Evaluated ......................................... 117
Table 6.3 - Optimal Mapping Scheme & Criteria ....................................................... 123
Table 6.4 - Implementation Results ............................................................................. 125
Table 6.5 - Performances for a 64x32x64 Backpropagation Neural System .............. 126
Table 6.6 - Performances for a 96x32x96 Backpropagation Neural System ................ 126
Table 6.7 - Performances for a 96x50x96 Backpropagation Neural System .............. 127
Table 7.1 - Neural Systems as Structured Information ............................................... 134
Chapter 1
Introduction

This chapter presents the motivations and aims of this thesis in decomposing and mapping neural systems on parallel architectures. A brief analysis of neural systems and how they relate to parallel computing is given. From this, the aims of this thesis are presented. Finally, the contents of the thesis are outlined.

1.1. Neural Computing

Neural computing attempts to mimic information processing and the cognition tasks of the brain via computer simulation [Ama77a]. The resulting computer models, called neural networks, are massively parallel interconnected networks of simple adaptive elements (neurons and connections) and their hierarchical organisations. These models are intended to process information in a "brain-like" manner in order to perform analogous functions, such as elementary memory functions [Koh87a]. The simulations of these models on sequential computers are computationally expensive. For example, a medium-size application, e.g. neural networks applied to exchange rates forecasting, would typically involve a sample size of 6000 to 10000 patterns. One single experiment requires, on a SUN workstation, approximately ten hours of CPU time [Ref92a]. Neural networks, however are inherently parallel; they involve large numbers of local computations that can, in principle, be carried out in parallel. The advent of parallel computers, such as the Transputer array [Kno89a], Connection Machine [Hil85a], or Distributed Array of Processors [Par90a] makes parallel implementations of neural networks possible. This thesis aims to analyse neural computing and the parallelism it embodies in order to exploit it uniformly across a range of general-purpose parallel machines.

1.1.1. Neural Network Systems

For the purpose of examining the parallelism of neural network systems, distinction is made between neural network models, neural network applications, and neural network systems. As we shall see later, neural networks exhibit parallelism at the model level and/or system level. A neural network is said to form a system made of a model, an application, and their integration. The following terms are thus defined: neural model, neural application, and neural system.
Figure 1.1 - Neural Network Model, Application and System

A neural network system is formed by the integration of a neural network model and application. The neural model is composed of a hierarchical set of neurons and connections, depicted within the square box. The neural application determines the input and output data, shown as $X, Y$ in the figure.

A neural network model is the description of the hierarchical organisation of the neurons, the network interconnectivity, the local operations of neurons, the local adaptation of connections, and their overall dynamic behaviour (i.e. learning). In other words, the neural model is the algorithm of a neural network system.

A neural network application is the description of the interactions between the model and its external environment. The model interacts with its environment via its input and output neurons, as shown in Figure 1.1. The input neurons receive inputs $X$ from the environment, and the output neurons provide outputs $Y$ to the environment.

A neural network system consists of the integration between a neural model and neural application. It specifies how the different elements forming the input $(x_1 \cdots x_n)$ and output $(y_1 \cdots y_m)$ interfere with the learning operation of the neural model.

The next sections present the basic elements of neural network systems, and a framework for describing them.

1.1.1. Basic Elements

Using computer science terminology, neural network systems are systems formed by a set of processing units, the neurons, interconnected according to a certain pattern, the network configuration, and which act together using weighted connections within an environment. A neuron and its weighted connections are the basic elements of neural network systems, as shown in Figure 1.2.
A neuron $i$, indicated by a circle in Figure 1.2, has a state of activation $x_i$ at each point in time. Associated with each connection there is a weight $w_{ij}$ which defines the effect of the unit $j$ on the unit $i$. The new activation $x_i$ at $t+1$ of a neuron is generally obtained by applying a function $f$ on the weights $w_{ij}$ and activation of neurons $j$ which connect to the neuron $i$. This function is referred to as activation rule and is of the form:

$$x_i(t+1) = f(x_i(t), \Sigma w_{ij} x_j)$$  \hspace{1cm} \text{Activation Rule} \hspace{1cm} (1.1)$$

The weights $w_{ij}$ undergo modification so as to construct an internal representation of their environment. The concept of learning is related to this adaptation of the weights, referred to as the learning rule [Hin87a]. The learning rule typically modifies the weights according to their previous value and the state or other variables associated with the neurons they are connecting.

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij} ; \Delta w_{ij} = g(w_{ij}, x_i, x_j, \cdots)$$  \hspace{1cm} \text{Learning Rule} \hspace{1cm} (1.2)$$

The weighted synapses (connections) between neurons specify the interactions that occur within a network and govern the influence of each particular neuron onto the others, i.e. the weights contribute to the input signals that a neuron receives. The modification of these weights over a time span represents the modification of the knowledge (i.e. learning) that a neural network exhibits. Taking these basic elements, i.e., neurons and connections, a neural network system is fully described by specifying its neurons and connections, the network connectivity pattern and its adaptation, and finally the interaction with the external environment, i.e. neural application. A framework for describing the range of neural network systems is now presented.

### 1.1.1.2. Framework

As indicated above, a neural system is the integration of a neural model and a neural application. A neural model has both local and global characteristics. Local characteristics are concerned with the individual or basic elements (neuron and connection) of the model whilst global characteristics are concerned with the configuration and the dynamic behaviour of the network of neurons and connections. A neural application defines the
function the neural model performs, it specifies the data to give to the network and the data to obtain from it. For example, in the case of a pattern associator, the application aims to retain a set of input-output pattern associations. Lastly, in order to integrate an application and a model, one has to specify how the operations of the model are to be organised and controlled, this is referred to as system processing because it is concerned with the high level transform performed by a model. Consequently, the description of neural network systems involves the specification of the model it uses, the application it intends to perform, and the associated system processing as summarised in the following table.

<table>
<thead>
<tr>
<th>Neural Network System</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Neural Model</strong></td>
<td></td>
</tr>
<tr>
<td>Local Components</td>
<td></td>
</tr>
<tr>
<td>Neuron's type</td>
<td>binary 0-1</td>
</tr>
<tr>
<td>Neuron's Activation Rule</td>
<td>( x_i = \frac{1}{1+e^{-z_i}} )</td>
</tr>
<tr>
<td>Connection's Adaptation Rule</td>
<td>( w_{ij} = w_{ij} + \Delta_{ij} )</td>
</tr>
<tr>
<td></td>
<td>( \Delta_{ij} = \alpha x_i x_j )</td>
</tr>
<tr>
<td>Global Components</td>
<td></td>
</tr>
<tr>
<td>Organisation of Network of Neurons</td>
<td>Layered</td>
</tr>
<tr>
<td>Network Interconnectivity</td>
<td>Full connectivity inter layers</td>
</tr>
<tr>
<td>Network Dynamic Processing</td>
<td>Adaptation of the weights</td>
</tr>
<tr>
<td><strong>Neural Application</strong></td>
<td></td>
</tr>
<tr>
<td>Data to the model</td>
<td>Patterns (e.g. set of pixels)</td>
</tr>
<tr>
<td>Data from the model</td>
<td></td>
</tr>
<tr>
<td><strong>System Processing</strong></td>
<td></td>
</tr>
<tr>
<td>Control of the Network Dynamic Processing</td>
<td>Batch learning of a set of patterns</td>
</tr>
</tbody>
</table>

**Table 1.1 - A Framework for Neural Network System**

A neural system is described by the model it uses, the application it performs, and the processing necessary for the achievement of the application. The neural model describes the behaviour of the network of neurons and connections, both locally and globally. The application specifies the data to give to and retrieve from the net. Finally, the system processing specifies the neural model processing to operate in order to perform the application.

As shown in Table 1.1, the specification of the model comprises local components and global components:

**Local components:**

Local components are the neurons and connections and their respective computations. A neuron is specified by its type, for example binary, and its computational or activation rule. This describes the neuron’s response to its input signal, and is a combination of the weights and neurons it is connected with, as described by Equation (2.1). Different types of neurons, both in terms of operations they perform and in terms of their type (binary or continuous) can coexist within a same model. Standard activation rules include functions such as step function, and sigmoids both symmetrical (\( \tanh \)) or asymmetrical (\( \frac{1}{1+e^{-x}} \)). A connection is specified by its adaptation or learning rule which describes the modification process of its weight value, i.e. the computation to perform to obtain \( \Delta w_{ij} \), such that \( w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij} \). Standard learning rules are typically functions of neurons’
activation state, error value, previous weight change, as described in Equation (1.2).

Global components

Global components are the organisation of the network of neurons, the interconnectivity of the network, and the network dynamic behaviour (i.e. processing of the model). The neurons' network organisation specifies how the neurons are (or not) hierarchically constructed, whilst the network interconnectivity describes the pattern of weighted connections which links neurons. For example, neurons can be organised in slabs with weighted connections linking neurons within different slabs. Moreover, the interconnectivity affects the way neurons interact with each other, i.e. neurons’ local rules are contingent to their weighted connections. The network dynamic behaviour specifies the processing occurring within the network. It describes how the local rules of the network’s elements operate and cooperate so as to achieve learning and storage. It specifies the processing order of neurons’ rules (e.g. on the basis of their dependence to a slab), and connections’ rules (e.g. on the basis of the neurons they are connecting).

The neural application is the description of the function the model intends to perform. This is complemented by the description of the data to give to, and obtain from the model. Then, once the application is defined, the overall processing, i.e. the processing required for the achievement of the application must be specified. This system processing indicates how the network dynamic behaviour (global component of a neural model) is to be controlled in order to perform the application. For example, in the case of a neural system used as a pattern associator, the neural model performs the learning of one pattern. The system processing specifies how this learning is controlled to achieve the learning of a set of patterns.

This section has presented the neural network paradigm and the resulting neural systems. It has also proposed a framework for their description which makes a distinction between neural model and neural application and system. The usage of neural network systems, i.e., the neural computing field is now examined.

1.1.2. Neural Computing Field

With regard to neural systems, the neural computing field can be divided into two basic areas:

- The study and application of neural network systems,
- The development of software and/or hardware tools to support this study and application.

The study and application of neural network systems comprises two groups of researchers and/or practitioners. Firstly, there are researchers whose prime interest lies in the development of neural models. Typically, the objective is to develop models with
enhanced convergence and generalisation properties. These new neural models need to be validated via simulations. Simple benchmarks are thus used, and the simulation results are analysed [Fal87a]. The properties performed by the neural network are compared with those expected by the theory to determine the correctness of the model. The whole process is largely experimental and is computationally intensive. Secondly, there are developers who build non trivial applications of neural models. Initially, they identify applications whose characteristics fit neural networks properties [Hop82a]. Then, because of the different properties available, developers experiment with various models in order to find the model with optimal convergence and generalisation, as shown in Figure 1.2.

Thus, for both researchers and practitioners, the investigation of neural network systems by means of software and hardware tools is of major importance. These tools allow the analysis and validation of new neural models, and the investigation of neural network capabilities for various applications. In recent years, quite a large number of such tools have been developed [Tre92a]. They range from educational tools to sophisticated neural programming environments. A good example of an educational tool is the Parallel Distributed Processing Environment (PDP) [Rum87a]. This environment provides a basic range of neural models and associated applications in order for its user to acquire knowledge in neural network systems. However, it gives no consideration to the parallel nature of neural systems. There also exist commercial neural systems tools, such as Nestor [Nestor]. However, they often lack flexibility in that they are generally application oriented. For example, Nestor is a pattern recognition environment. Lastly, there are more complex environments for neural networks which may provide both software and hardware tools. A typical example is the ANNE environment which comprises a network description and intermediate level language, and a mapper with a physical architecture descriptor to partition the network amongst iPSC processors [Bah88a]. Our interest lies in environments that provide mechanisms for exploiting the parallelism exhibited by neural systems. A detailed review of neural environments can be found in [Tre89a].

1.2. Parallel Computing

This thesis studies the parallel computing domain with respect to the neural computing field. Thus, the relations between neural computing and parallel computing are now examined.

1.2.1. Neural Computing & Parallel Computing

Neural systems are inherently parallel, and their study and application is experimental, i.e. it involves a large number of simulations. High-performance machines are thus seen as the enabling technology for developing neural network systems. There are three
groups of high-performance machines which support neural computing:

- co-processor accelerators,
- neurocomputers,
- general-purpose parallel machines.

Co-processor accelerators are typically single boards that plug into the backplane of an IBM PC, or interface to a SUN Workstation. These boards might contain, for example a Motorola MC68020 plus a floating-point co-processor together with a large memory for implementing the neurons and connections [Tre90a]. They accelerate specific operations performed by a neural network simulation. However, they are inherently sequential and thus they can only partially exploit the potential for parallelism of neural networks.

Neurocomputers are the result of the investigation of new architectural designs for building hardware tools dedicated to neural network simulation [Tre89b]. The work done in this area can be divided into two parts: i) the development of specialised VLSI neuro-chips, and ii) the development of general-purpose neurocomputers. Specialised VLSI neuro-chips are often application-specific, and are developed to speed up a specific neural network, for example the Kohonen neural network. In this respect, they provide good performance but lack flexibility. On the other hand, general-purpose neurocomputers are generic in that various neural networks can be implemented on them. For example, Adaptive Solution Inc [Sol91a], has developed neuron-based neurocomputer, named CNAPS, Connected Network of Adaptive Processors. This neurocomputer is optimised for traditional neural network systems, e.g. fully connected networks, but is general enough to support more complex neural systems [Hec90a]. However, the usefulness of developing dedicated hardware is debatable, and its cost/effectiveness has yet to be validated [Ref90a].

General-purpose parallel machines are becoming increasingly available, and are seen as a cost effective alternative because they provide an execution environment for virtually any type of application. The same machine can be used to support the parallel implementation of an image processing application [Ara86a] as well as a data base application [Pog88a]. The work done so far in exploiting neural networks parallelism on these machines is promising; substantial speedups have been obtained [Zha90a]. However, the software tools available are not generic in that they do not cover a spectrum of machines. In this thesis, we address the generic problem of decomposing and mapping neural networks onto general-purpose parallel machines. In this area, the existing work can be divided into four categories, ranging in complexity from the implementation of a specific neural network system on a specific machine, to the provision of an environment that maps a range of neural systems onto a range of machines. They can be characterised as follows:
• One-to-One: mapping the parallelism of a particular neural system onto a particular parallel machine [Ric89a],
• Many-to-One: mapping a range of neural systems onto a particular parallel machine [Mig90a],
• One-to-Many: mapping a specific neural system onto a small set of different parallel machines [Obe90a],
• Many-to-Many: mapping a range of neural systems onto a range of parallel machines.

The work done in this thesis falls into the last category: the many-to-many mapping. That is, for virtually any neural system the aim is to develop a mechanism able to produce an efficient mapping onto a range of general-purpose parallel machines. Hence, the objectives are the expression of parallelism across a range of neural systems, and the exploitation of parallelism across a range of parallel machines. Before specifying the objectives further, the range of parallel machines considered in this thesis is reviewed.

1.2.2. Parallel Machines

The purpose of this section is to classify parallel machines according to their architectures in order to identify which class of parallel machines this thesis addresses. A wide variety of parallel architectures has been proposed. Attempts to classify their designs unequivocally have somewhat failed, and as such there are no complete and accepted classification schemes [Hoc84a]. Classification schemes have shown their limitations in that there will always be a particular parallel architecture that will not fit into the classification. However, these classifications are recognised by the whole parallel computing community, and provide useful tools for expanding our understanding of parallel processing and give a background analysis for decomposition strategies.

Firstly, the Flynn’s classification [Fly66a] considered as a reference, is based on the relations between the instructions and the data processed by a machine. It defines the concept of streams (of either data or instructions) and classifies machines by determining whether the streams are single or multiple. From this, four broad machines’ types are defined: i) SISD Single Instruction Single Data, ii) SIMD Single Instruction Multiple Data, iii) MISD Multiple Instructions Single Data, and iv) MIMD Multiple Instructions Multiple Data. Obviously, because the SISD type of architecture defines purely sequential machines, it is of no interest within the scope of this thesis. The SIMD class defines machines whose multiple processors simultaneously execute a same instruction on different data streams. The MISD class defines machines whose multiple processors apply different instructions on the same data streams. The practical realisation of such a type has been ruled out. The MIMD class defines machines whose multiple processors
execute autonomously on different data streams. This classification is a useful shorthand for characterising different architectures [Dun90a]. It is however incomplete. Pipelined vectors processors, such as the Cray [Hwa84a], recognised as parallel architectures (they allow concurrent arithmetic execution), do not belong to any of the four classes [Hoc87a]. They are not SIMD because there are no processors that execute the same instructions, they are not MIMD either because processors have no autonomy as such.

Secondly, there exist recent classification schemes which attempt to merge the properties and components that have been defined as characteristics of parallel architectures in previous schemes. Duncan provides a high level classification aiming to encompass all existing parallel architectures [Dun90a]. This distinguishes three high level classes:

- the synchronous architectures,
- the MIMD architectures,
- the MIMD-based architectures.

They are summarised in Figure 1.3.

![Parallel Architectures Diagram]

**Figure 1.3 - Classification of Parallel Machine Architectures**

The Synchronous class incorporates the vector processors, the processor array and associative memory (part of the SIMD Flynn's class), and the systolic architectures. The MIMD class incorporates distributed memory and shared memory systems. The MIMD-based architectures incorporate the dataflow architecture, the reduction and the wavefront architectures.

**Synchronous architectures**

The synchronous class is composed of parallel architectures that operate concurrently in lockstep through either global clocks, central control unit, or vector unit controllers. It is further divided into three sub-classes: the vector architectures, the SIMD architectures, and the systolic architectures.

The vector architectures are characterised by pipelined multiple arithmetic units that operate concurrently. They provide parallel vector processing by "chaining" results of units into inputs for the following units. The Cray architectures are typical examples [Hwa84a].

The SIMD architectures employed a central control unit, multiple processors and an interconnection network for either processor-processor or processor-memory communication. This class incorporates both the processor arrays and associative memory.
architectures. The Distributed Array of Processors (DAP) is a typical example of processors array, while the PEPE architecture is representative of the associative memory architectures [Par88a].

The systolic architectures were initially built to solve the problems associated with demanding Input/Output (I/O) bandwidth systems. They are pipelined multiprocessors in which data is fed into the processors in rhythmic fashion; a datum enters the systolic array and is then passed to any processor that requires it. Thus, only boundary processors perform I/O operations to and from the memory. The Warp is an example of reconfigurable systolic architecture [Kun87a].

**MIMD architectures**

The MIMD architectures are architectures composed of a set of processors that can execute independently using local data. They are asynchronous with decentralised hardware control; their synchronisation is effectively performed either by message passing through an interconnection network or by accessing data in shared memory units. Thus, this class comprises distributed memory architectures and shared memory architectures.

In the case of distributed memory MIMD, processing nodes; i.e. processor + local memory, are connected with processor-to-processor interconnection network. There exist a wide variety of interconnection networks, each of which defines a different interprocessor communication pattern. The typical ones are the ring interconnection network where nodes form a ring; the mesh interconnection network where individual nodes are connected to their four immediate neighbours; and the hypercube interconnection network which arrange the nodes so to form a n-dimensional cube. The Parsys and Ncube are typical examples of this class.

The shared memory MIMD architectures involve multiple processors sharing memory. The access to shared memory units acts as the synchroniser. In these architectures, processors are general purpose rather than just composed of a CPU and peripheral I/O processors. An interconnection network is provided by the architecture to support memory access. For example, the Alliant FX/8 uses a crossbar scheme to connect processors and cache memories [Per86a]. The crossbar prevents contention for communication links by providing a dedicated path way between each possible processor/memory pairing.

**MIMD-based architectures**

This class incorporates all hybrid parallel architectures; it is composed of architectures that might be based on the MIMD paradigm but whose organising principle differs from the MIMD. Hybrid MIMD/SIMD, dataflow architectures, and reduction architectures composed this class.
The hybrid MIMD/SIMD architectures are partly MIMD and partly SIMD; i.e. a group of its processors are independent and "obey" MIMD rules, while another set is globally controlled (for example, by one of the MIMD processors). The Texas Reconfigurable Array Computer is an example of such an hybrid parallel architecture [Lip87a].

The dataflow architectures are based on data dependencies; that is, executions of instructions occur as soon as their operands are available [Tre82a]. The Manchester Dataflow computer is an example of the practical realisation of the dataflow principle.

The reduction architectures are demand-driven architectures, and are based on data requirements. That is, an instruction is enabled if its results are required by another enabled instruction. Historically, they have been developed to provide architectural support for functional languages.

Lastly, it is worth mentioning the wavefront array architectures which are a combination of systolic data pipelining and asynchronous dataflow execution principle. These architectures were developed for special-purpose systems that balance intensive computations with I/O bandwidth. They are characterised by modular processors and local interconnection networks with handshake mechanism that sequences the computations. The John Hopkins university built such a parallel architecture, however these architectures are still in an evaluation state.

In summary, the above classification groups architectures in three main classes: synchronous, MIMD, and MIMD-based. The MIMD-based class incorporates all unclassifiable architectures; e.g. the hybrid MIMD/SIMD. This classification permits us to identify some basic parameters which define parallel architectures such as, for example the process synchronisation (e.g. synchronous, asynchronous, hybrid), and inter-processor communication (e.g. interconnection network, array-based). More importantly, it allows the specification of the type of parallel architectures this thesis considers. Both the basic synchronous and MIMD architectures are targeted in this thesis, this covers quite a wide spectrum of alternative parallel architectures. Moreover, the hybrid architectures because combining SIMD and MIMD characteristics is also part of our universe of parallel architectures. Finally, it shall be noted that architectures which allow Input/Output operations and computations to occur in parallel are not considered as parallel.

After this introduction to the neural computing field and its relations to the parallel computing field, the motivations and aims of this thesis are presented.

1.3. Thesis Motivations & Aims

As mentioned earlier, this thesis addresses the problem of decomposing and mapping neural networks onto general-purpose parallel machines. This problem deals with two difficulties:
• There are only a few languages dedicated to neural networks which express their parallelism,

• There are no generic systems or tools which enable the exploitation of the parallelism in a range of neural networks across a range of general-purpose parallel machines.

Initially, the neural computing community focused on the development of neural programming environments, ranging from educational environments such as the PDP environment [Rum87a], to commercial environments such as the Nestor Development System [Nes88a]. However, these environments are largely dedicated to sequential machines [Aze92a]. Similarly for the representation and programming of neural networks, i.e. few specifications provide methods for expressing their parallelism. Firstly, because neural environments are executable only on sequential machines, parallelism is not considered as an issue, and thus specifications do not support parallelism. Secondly, environments that execute on specific parallel computers provide specifications for neural networks which support the parallelism of the targeted machine, for example the CARELIA environment [Koi89a]. Furthermore, once a working neural application is constructed, its specification is often tailored for implementation on a specific parallel machine, i.e. dedicated parallelisation. For example, Zhao in [Zha90b] describes the implementation of the Traveling Salesman Problem on a Meiko parallel computer. Consequently, the work done has not provided any kind of formal or in-depth investigation of the parallelism of neural networks.

This thesis aims:

• To analyse and extract the parallelism found in neural networks,
• To exploit it across a range of general-purpose parallel machines.

The ultimate goal is to construct a mapping software that exploits the intrinsically parallel nature of neural networks across a range of conventional (non dedicated) parallel machines. This mapping software adapts jointly to the neural network and its parallelism and to the parallel machine. It analyses and extracts the parallelism of any neural network and deduces a mapping which is the most suitable for the type of parallelism of the targeted machine.

To do so, this thesis proposes a Mapping System for decomposing and mapping neural networks onto general-purpose parallel machines.
As shown in Figure 1.4, the System divides the mapping task into two modules: a neural Abstract Decomposition and a Machine Dependent Decomposition. The AD is responsible for extracting the parallelism that neural systems exhibit and for identifying the associated alternative decomposition schemes. The MDD considers the targeted parallel machine and produces the specification of the neural systems onto the machine. The System provides a framework for decomposing and mapping neural networks which aims to combine genericity with efficiency, i.e. enabling the mapping of virtually any neural network onto virtually any parallel machine while optimising the exploitation of the available parallelism.

1.4. Thesis Outline

This thesis is divided into nine chapters. This chapter has introduced the neural computing research field, its relations with the parallel computing field, and has explained the aims of the research.

Chapter 2 firstly reviews the work done in the area of neural systems and parallel machines. This reveals weaknesses in current systems for describing and exploiting the parallel nature of neural networks. Secondly, in order to have a common set of references, this chapter surveys both neural network systems and general-purpose parallel machines.

Following from this, Chapter 3 investigates the parallelism of both neural systems and parallel machines. Parallel machines and the approaches to parallel programming are analysed, and the parallelism of neural systems is classified in accordance with their framework. This makes it possible to identify the requirements for a Mapping System for decomposing and mapping neural networks onto general-purpose parallel machines.
Chapter 4 proposes a *Mapping System* for decomposing and mapping neural network systems. The *Mapping System* divides the mapping task into two: a neural system Abstract Decomposition (AD) and a Machine Dependent Decomposition (MDD). The Abstract Decomposition analyses the parallelism exhibited by neural systems and specifies possible decomposition schemes. The Machine Dependent Decomposition deduces a suitable neural system's mapping scheme according to the targeted parallel machine.

Chapter 5 describes the construction of a prototype system that automatically maps neural systems onto a Transputer-based machine. The mapping software has been developed within the Esprit project Pygmalion [Tre89c], and is targeted to Transputer-based machines which are representative of the MIMD machines. It implements an automatic translation from a neural network specification to its equivalent on a Transputer-based machine, namely the Parsys SN1000.

Based on the *Mapping System* proposed in Chapter 4 and on its prototype implementation described in Chapter 5, two major improvements are made. The first one concerns the optimisation of the mapping process and is presented in Chapter 6, while the second one concerns the optimisation of the expression of neural systems parallelism and is presented in Chapter 7.

Chapter 6 optimises the exploitation of parallelism by developing an analytical framework for forecasting the speed-up of neural systems mapping schemes. This additional tool is able to select a particular neural network mapping scheme according to its predicted performances. The usage and validity of this tool is demonstrated for an SIMD machine, namely the Distributed Array of Processors.

Chapter 7 focuses on the expression of neural systems parallelism. It introduces a specification for neural systems with explicit support for parallelism. The specification uses the *Abstract Syntax Notation One* (ASN.1) as the syntactic constructs. The interest of this work resides in the innovative use of ASN.1 for specifying neural systems; i.e. so far, ASN.1 has been dedicated to the specification of communication protocols.

Chapter 8 evaluates the thesis’s research, covering five areas of work. Firstly, the analysis of parallelism in neural systems and machines is assessed. Secondly, the *Mapping System* and its two modules; the AD and MDD, are assessed. Thirdly, the MIMD prototype is evaluated, and compared with existing mapping softwares. Fourthly, the analytical framework and the experiments carried out on an SIMD machine are evaluated. Lastly, the specifications for neural systems: nC and ASN.1-based are assessed and compared.

Finally, Chapter 9 assesses the research conducted and presents the dissertation’s conclusions.
1.5. Thesis Contributions

This thesis develops a solution for decomposing and mapping neural systems onto general-purpose parallel machines. In doing so, this thesis makes three major contributions: i) a general-purpose Mapping System for mapping neural systems onto parallel machines, ii) an analytical framework for evaluating the speedup of alternative neural system mappings, and iii) an ASN.1-based specification for neural systems with explicit support for parallelism. Based on this, the research reported in this thesis can be divided into five areas:

- The analysis of parallelism in both neural systems and parallel machines. This systematic analysis demonstrates the need and identifies the requirements for a mechanism for expressing and exploiting the parallelism of neural systems onto parallel machines.

- The development of a Mapping System for decomposing and mapping neural systems onto general-purpose parallel machines. This Mapping System is applicable to virtually any neural system and parallel machine.

- The implementation of a mapping software for neural systems onto a typical MIMD machine. This mapping software, developed as part of the Esprit Pygmalion Neurocomputing project, automatically decomposes neural systems, and generates their parallel specification on the 48-processor Parsys SN1000 Transputer-based machine.

- The development of an analytical framework for evaluating the speedup of alternative mapping schemes of neural systems onto parallel machines. The validity of this additional tool is demonstrated using an SIMD machine; the Distributed Array of Processors (DAP).

- The innovative use of the Abstract Syntax Notation One (ASN.1) as the notational support for specifying neural systems with explicit support for parallelism.
Chapter 2
Neural Network Systems & Parallel Machines

This chapter firstly reviews the work done in the area of mapping neural network systems onto parallel machines. Secondly, it presents some typical examples of neural systems and parallel machines, and derives a taxonomy for both of them.

2.1. Related Work

As indicated in Chapter 1, the objective of this thesis is to develop mechanisms for analysing and exploiting the parallelism of neural network systems across a range of parallel machines. Thus, to motivate the work done in this thesis, the various approaches in dealing with the mapping of neural systems onto parallel machines are reviewed. Four kinds of mappings are identified:

- **One-to-One:** mapping the parallelism of a particular neural system onto a particular parallel machine [Ric89a], and [Zha90b], and [Bod90a], and [Wit89a].

- **Many-to-One:** mapping a range of neural systems onto a particular parallel machine [Mig90a], and [For87a], and [Em90a].

- **One-to-Many:** mapping a specific neural system onto a small set of different parallel machines [Obe90a], and [J.B90a].

- **Many-to-Many:** mapping a range of systems onto a range of parallel machines.

The one-to-one category investigates different methodologies for distributing a specific neural system onto specific hardware. These studies are useful background; they give descriptions of particular neural systems decomposition according to the targeted hardware, they are often the most efficient. Typically, they present an evaluation of load balancing and routing according to the topology of the system on the machine. In the case of coarse grain target hardware, they give a detailed description of various neuron-to-processor allocations, including an evaluation of their computational vs communication cost. Moreover, this first category provides us with various data distributions, according to the system studied. For example, Richards describes a particular geometric decomposition of the connectivity matrix which speeds up the simulation for a sparsely connected backpropagation neural network on a Transputer-based machine [Ric89a]. Zhang details the implementation of a backpropagation system on the Connection Machine [Zha90a]. In this case, the benefit of a shared memory in the machine is shown; i.e. it allows the exploitation of a specific aspect of backpropagation (weights of connecting layers can operate in parallel). However, solutions found in this line of development are too specific when one tries to exploit
neural parallelism on a range of parallel machines.

The many-to-one category consists of a machine-specific strategy for mapping aspects of neural systems [For87a]. Specific features of the target machine are provided in a software framework. Furthermore, these frameworks specify the means for data placement, communication schemes, control of the neural system processing, etc. More specifically, they present approaches for specifying neural systems that support the expression and exploitation of parallelism. The Carelia simulator [Koi89a] describes a neural networks development environment designed to run in a multi-transputer processor network. Within this environment, the neural systems are expressed using a Communicating Sequential Processing (CSP) formalism [Hoa78a]. Thus, the basic features of the specification, the approach used to describe neural systems, makes the exploitation of their parallelism strongly biased towards CSP-based machines, i.e. message-passing multicomputers. In conclusion, this category is targeted to a specific machine, whereas this thesis intends to consider a whole range of machine-specific elements that intervene when mapping neural systems.

The one-to-many category includes mapping a specific neural system onto a range of parallel machines. This allows, for example, comparisons to be made between different machines. In [Obe90a], the simulation of a self-organising system onto a Transputer ring and a Connection Machine CM-2 are compared. The parallelism exhibited by the self-organising feature maps; developed by Kohonen [Koh87b], is analysed. Then, its exploitation on both machines is described, and their respective performances presented. The performances are expressed as a function of machine's parameters, it is therefore useful because important machine criteria are thus identified. In this case, different criteria are devised for the different machines; giving thus some information about which criteria are crucial for which parallel machine. However, firstly this category is restricted to particular neural system, and secondly does not provide a 'unified' framework for the performances of each implementation (often, criteria are machine dependent, thus too specific to be applied to a range of machines).

Considering now the many-to-many category, there is a lack of support of the parallelism exhibited by neural systems, both in the expression and exploitation of parallelism. Indeed, investigating the literature one soon realises that there are no specifications that express the parallelism in a way suitable to allow a mapping onto different types of parallel machines. Furthermore, there are no systems sufficiently generic to support their execution (implementation) on a range of parallel machines. The aim of this thesis is to develop a many-to-many system. This has proved an impossibly difficult problem for general-purpose tasks but, as we shall see next, it is eased by considering this problem in the context of neural systems.
Before focusing on the parallelism of neural network systems and parallel machines, it is useful to provide a taxonomy of both neural systems and parallel machines; this allows us to have a common framework for references.

### 2.2. Neural Network Systems

Based on the framework presented in Chapter 1, this section describes typical examples of neural network systems which have been selected to form a representative range of their abilities and characteristics. It permits us to present a taxonomy of neural network systems and also to have a basic set of neural systems for reference when analysing parallelism.

#### 2.2.1. Survey of Neural Network Systems

Within the literature, two main classes of neural systems are defined: *supervised* systems and *unsupervised* systems. Supervised systems use neural models in which the external environment, i.e. the application dictates the input and output of the model. The objective of such systems is thus to *learn* input-output associations. On the other hand, unsupervised systems are "free" systems; they produce their own response to input from the environment. Another class of neural systems is worth mentioning: the *combinatorial optimisers* which are able to solve combinatorial optimisation problems. In this case, it is the state of the neural system itself that represents the solution [Aar89a].

The neural systems described below are representative of each class. These are respectively: the Hopfield neural system as a combinatorial system, the backpropagation as a supervised system, and the Self Organising Maps as an unsupervised system. Their description follows the framework introduced in the previous section. The neural models are determined in terms of local and global components, and an example of their integration with an application is given. The following notation is used:

- \( x_i \) = activation state for neuron (unit) \( i \)
- \( d_i \) = desired state (output) for neuron \( i \)
- \( e_i \) = error term for neuron \( i \)
- \( w_{ij} \) = connection weight from neuron \( j \) to neuron \( i \)
- \( \theta \) = a threshold value
- \( \lambda \) = a scaling factor, referred to as learning rate

#### 2.2.1.1. Combinatorial Optimiser: Hopfield

The Hopfield neural system originates from spin-glass theory and was initially described by Hopfield in [Hop82a]. Related systems include the Boltzmann machine which is based on an analogy between statistical mechanics and combinatorial...
optimisation [Aar87a].

**Neural Model**

The Hopfield model is made of binary neurons (cf +1/-1 spins) with fixed connections. The connections take real values of either sign and are set to embed the constraints of the optimisation problem. Furthermore, the connections are symmetric (\( w_{ij} = w_{ji} \) if \( i \neq j \), \( w_{ii} = 0 \)). Neurons obey the following activation rule:

\[
\begin{align*}
    x_i &= 1 \text{ if } \sum_j w_{ij} x_j > \theta \\
    x_i &= 0 \text{ otherwise}
\end{align*}
\]

*Hopfield Activation Rule*

The model is made of one set of neurons with full connectivity in between. Because the connections are fixed, the model performs one simple computation referred to as *relaxation* computation, in which neurons evolve by computing their activation state. It should be noted that neurons can compute their new state in parallel.

**Neural Application & System**

The application sets the connections weights to represent the constraints of the optimisation problem. For example, in the case of Hopfield applied to the Travelling Salesman Problem the weights are set so as to ensure that each city is traversed only once [Aar89a]. The computations performed by the Hopfield system simply leaves the model to evolve until the model stabilises. It has been shown that ultimately, the network of neurons (the model) converges to a stable state which corresponds to a minimum point of an *energy* function of the form:

\[
E = -\frac{1}{2} \sum_{ij} w_{ij} x_i x_j.
\]

### 2.2.1.2. Supervised Systems: Backpropagation

The hidden components of a neural system are not directly exposed to the external environment. The problem of how to govern their influence initiated research [Wer88a], [Par85a], and [Rum87a], and resulted in the description of a neural model aiming to give better control over the knowledge encoded by the hidden components. The basis is to compute a measure of the error; the learning process is then about reducing this error measure. The backpropagation system described here is referred to, in the literature, as the *standard* backpropagation [Rum87a]. Multiple variations of this system exist, they are however not considered here.

**Neural Model**

---

1 These variations of the backpropagation system are of little interest for this section which describes typical neural systems to obtain an overall view of their abilities and types of processing.
Neurons take real values and perform two rules; the activation rule and a rule for computing the neurons' local error. The activation rule, which determined the actual activation state of a neuron, is a non-linear function of the neurons input signals, and is typically of the form:

\[
x_i = 1 + e^{-x_i} w_i x_i
\]

**Backpropagation Activation Rule**

For the computations of the neurons' error, distinction is made between the visible neurons and the hidden neurons. The error for an hidden neuron is determined recursively in terms of the error signals of the neurons to which it connects \( \delta_j \), and the weight of these connections \( w_{ji} \). For an output neuron, the error is based on the difference between the desired state of the neuron \( d_i \) and its actual state \( x_i \). Formally, this gives:

\[
\delta_i = (d_i - x_i) x_i (1-x_i) \quad \text{Output Neuron}
\]

\[
\delta_i = \Sigma_j (w_{ji} \times \delta_j) x_i (1-a_i) \quad \text{Hidden Neuron} \quad \text{Backpropagation Error Computation}
\]

Weights are modified according to the derivative of a global error measure \( E \) along \( W \) (set of weights) as defined above. This gives:

\[
w_{ij} = w_{ij} + \Delta w_{ij}
\]

where \( \Delta w_{ij} = \lambda \delta_i a_j + \alpha \Delta w_{ij} \)

**Backpropagation Learning**

The neurons in the backpropagation model are arranged in layers - there are an input layer, an output layer and one or more hidden layers. Adjacent layers, visible or not, are fully connected. The model performs two global computations: learning and recalling. Learning trains the model for an application-specified entity; referred to as training pattern and made of an input-output pair. Recalling retrieves the output given its associated input. The model learning computation divides into three phases: a forward phase where each neuron computes its new state, a backward phase where each neuron computes its error and an update phase where each connection adapts its weight. These phases are computed sequentially. The forward phase starts from the first hidden layer until the output layer. All neurons belonging to the same layer can compute their activation state in parallel. The backward phase starts from the output layer and until the first hidden layer. As for the forward phase, all neurons of a single layer can compute their error in parallel. Lastly, the weights are updated according to the learning rule which then reduces the overall error measure. The whole set of connection weights, i.e. connections linking any neuron, can be updated in parallel. The recall computation is simply a forward phase, i.e. an input is presented to the model and is propagated through until the output neurons produce their response.
Neural Application & System

The computations performed by backpropagation systems aim to train the model (and recall) on a set of patterns, as defined by the application. The training involves the presentation, in turn, of each training pattern. If the set of training patterns $T$ is defined such that: $T = t_1, ..., t_i, ..., t_T$, the system processing consists of training the model for each $t_i \in T$, this is repeated for the whole set $T$. Indeed, when the model is trained for a pattern $t_i$ such that $i > k$ then the model enforces the representation of the pattern $t_i$ within the model, and thus reduces the representation of the patterns $t_k$. In consequence, the system has to process the whole set of patterns more than once. Furthermore, it is possible to learn each training pattern independently, this allows the system to operate in parallel over the training patterns [Par90b].

The overall processing of both optimisation and supervised neural network systems is driven by the application. There exist neural systems which produce their own response from the environment: the unsupervised systems.

2.2.1.3. Unsupervised Systems: Self Organising Map

Unsupervised neural systems are systems that self organise and produce their own response to their environment. For any type of neural system, supervised or not, it is the specification of the dynamic behaviour of the system’s elements that controls its response. Examples of such neural systems are Self Organising Map system [Koh87b], Competitive Learning [Rum87a], and Adaptive Resonance Theory (ART) [Gro88a]. The Self Organising Map system (SOM) is based on the assumption that the brain realises a topology-preserving mapping from the sensory environment to sensory-specific areas, i.e. the brain forms topographically ordered maps which are assumed to organise themselves.

Neural Model

In the SOM model, both neurons and connections take real values. Neurons, referred to as topological neurons compute their distance to an input vector relative to their weighted connections. The computation of this distance measure is:

$$d_i = \sum_j |x_j - w_{ij}| \text{ SOM Distance Computation}$$

The neuron with the minimal distance (min$_i (d_i)$), and its neighbours, $x_j \in N_i$ adapt their weights. Weights connecting neurons that do not respond to the input data (not minimal or neighbour) stay unchanged. Formally, for a neuron $k$, it gives:

$$w_{kj}(t+1) = w_{kj}(t) + \lambda (x_j - w_{kj}(t)) \text{ if neuron } k \text{ in neighbourhood } N_i$$

$$w_{kj}(t+1) = w_{kj}(t) \text{ otherwise SOM Learning Rule}$$
The SOM model is organised in two layers with full connectivity. The first layer acts as a relay layer, while the second layer acts as the topological map. The SOM model performs two global computations: clustering and matching. Clustering involves the identification of the winner neuron and the adaptation of its own weights and the weights of its neighbour. Matching simply involves the identification of the winner neuron. Firstly, to identify the winner neuron, neurons in the first layer propagate the input; they transmit their value directly to the second layer of neurons. Each neuron within this layer can transmit in parallel. Secondly, all topological neurons compute their distance, and the one with the minimal distance is marked. The computations of distance by the neurons can be performed in parallel. Once a neuron is declared winner, its neighbourhood $N_i$ is identified. $N_i$ is defined such that all neurons which lie within a certain radius from neuron $i$ are included within. Kohonen showed that best results in self organisation are obtained if the topological neighbourhood is a function of a discrete time index $t$ such that $N_i$ is selected fairly wide at the start and then slowly decreases with time [Koh87b]. The update phase requires the weights connected to the winner neuron and its neighbours to update their value according to the SOM learning rule, as defined above.

**Neural Application & System**

SOM is generally applied to classification problems. An SOM system takes a set of input data, defined by the application, and processes each data item through the clustering process of the SOM model. By repetitive iterations of this process input data are finally clustered. Even though an SOM system processes a set of patterns, it is not possible to learn each pattern independently (and thus in parallel), as in backpropagation systems.

### 2.2.2. Taxonomy

The set of neural systems presented above is far from exhaustive; small variations of one element of a system lead to a different one. For example, the connectivity pattern of the backpropagation neural model can be modified so as to use shared weights (two sets of neurons in adjacent layers share the same set of connections). This type of backpropagation model is used in pattern recognition and performs a windowing of the input images [Fal87a]. Shared weight backpropagation models support shift invariance properties. The neural systems presented are representative of the main types of neural systems; e.g. supervised and unsupervised, layered and unlayered. Table 2.1 summarises the features common to groups of neural systems, and details each of their components one step further.
Table 2.1 - Neural Network Systems: Common Features and Examples -

Any neural system is formed by the integration of a neural model with a neural application. Any neural model is determined by its configuration, and its processing. Its configuration determines the organisation of the neurons (for example organised in layers), and the interconnectivity pattern. It processing is divided into global (network) processing and local processing (neurons and connections). An application defines the data given to and retrieved from the model. A neural system processing defines the series of network processing to operate in order to perform the application.

Table 2.1 displays the features common to all neural systems, and gives an example of their values for the Hopfield, Backpropagation and Self Organising Map neural systems. Neural systems are composed of a neural model, an application and the processing to operate to perform the application. For example, the Hopfield neural system applies to optimisation applications, as shown in Table 2.1, and operates a simulated annealing process to achieve the optimisation (referred to as annealing in the table). A neural model provides configuration and processing information. The configuration describes the organisation of the neurons and the connectivity pattern, for example layered and full in the case of backpropagation in the Table 2.1. The processing specifies the computations performed by the network, referred to as global in the table, and by the neurons and connections, referred to as local in the table. For example in Table 2.1, the Self Organising Map model performs two global/network computations: winner which extracts the winner neuron of the clusters, and weight update which adapts the connection weights. In this case, local processing refers to the standard computations performed by the neurons and connections.

As we have seen in the case of connectivity of the backpropagation neural model, each of these features can vary. Each variation leads to the specification of a new neural model. Therefore, there exists a wide range of neural systems but they are all composed by common basic features which express their properties (e.g. configuration), and their processing (e.g. learning operation). These commonalities are further explained in Chapter 3 and are used when neural systems and the different types of parallelism they exhibit is discussed in section 3.2. Before doing so in the next chapter, general-purpose

<table>
<thead>
<tr>
<th>Example</th>
<th>Neural Network System</th>
<th>Neural System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Neural Model</td>
<td>Processing</td>
</tr>
<tr>
<td></td>
<td>Configuration</td>
<td>Neurons</td>
</tr>
<tr>
<td>Hopfield</td>
<td>Non-layered, Full &amp; Symmetric</td>
<td>evolve</td>
</tr>
<tr>
<td></td>
<td>Backpropagation</td>
<td>Layered, Full or Sparse</td>
</tr>
<tr>
<td></td>
<td>Self Organising Map</td>
<td>Layered, Full</td>
</tr>
</tbody>
</table>

Any neural system is formed by the integration of a neural model with a neural application. Any neural model is determined by its configuration, and its processing. Its configuration determines the organisation of the neurons (for example organised in layers), and the interconnectivity pattern. It processing is divided into global (network) processing and local processing (neurons and connections). An application defines the data given to and retrieved from the model. A neural system processing defines the series of network processing to operate in order to perform the application.

Table 2.1 displays the features common to all neural systems, and gives an example of their values for the Hopfield, Backpropagation and Self Organising Map neural systems. Neural systems are composed of a neural model, an application and the processing to operate to perform the application. For example, the Hopfield neural system applies to optimisation applications, as shown in Table 2.1, and operates a simulated annealing process to achieve the optimisation (referred to as annealing in the table). A neural model provides configuration and processing information. The configuration describes the organisation of the neurons and the connectivity pattern, for example layered and full in the case of backpropagation in the Table 2.1. The processing specifies the computations performed by the network, referred to as global in the table, and by the neurons and connections, referred to as local in the table. For example in Table 2.1, the Self Organising Map model performs two global/network computations: winner which extracts the winner neuron of the clusters, and weight update which adapts the connection weights. In this case, local processing refers to the standard computations performed by the neurons and connections.

As we have seen in the case of connectivity of the backpropagation neural model, each of these features can vary. Each variation leads to the specification of a new neural model. Therefore, there exists a wide range of neural systems but they are all composed by common basic features which express their properties (e.g. configuration), and their processing (e.g. learning operation). These commonalities are further explained in Chapter 3 and are used when neural systems and the different types of parallelism they exhibit is discussed in section 3.2. Before doing so in the next chapter, general-purpose
parallel machines are surveyed.

2.3. Parallel Machines

This thesis considers what is often referred to as general-purpose parallel machines, in the sense that they are accepted as workable general-purpose parallel computers [Bel92a]. The purpose of this section is to identify the parameters that characterise such a machine; this is necessary for developing a System for decomposing and mapping onto general-purpose parallel machines. The Chapter 1 has presented parallel architectures, this section concentrates onto parallel machines; i.e. their architecture and software environment.

2.3.1. Survey of Parallel Machines

Parallel machines may be defined as the combination of an architecture and software tools to support their exploitation. A possible definition is as follows:

*Parallel machines provide an explicit framework for the development of parallel programming solutions by providing multiple processors whether simple or complex, which cooperate to solve problems through concurrent execution.*

Parallelism is sometimes referred to as the execution of a concurrent model. In this case, concurrency is thus defined as the abstract property leading to parallel execution. The need to enforce such definitions in this thesis is seen as adding complexity rather than information, and is thus not enforced. Three typical parallel machines, their architecture and their software support are now presented. These machines are representative of various architectures: the Parsys SN1000 machine is a MIMD distributed memory machine [Par89a], the Connection Machine is a SIMD configurable machine [Hil85a], and finally the DAP, standing for Distributed Array of Processors is a SIMD processor-array machine [Par88a].

2.3.1.1. MIMD Distributed Memory: Transputer-based Machine

Transputer-based machines correspond to a realistic implementation of the Multiple Instruction Multiple Data (MIMD) concept associated with a pure distributed memory model. Their design is based on the theory of Communication Sequential Processes (CSP) introduced by Hoare [Hoa78a]. CSP defines sequential processes that can transfer information between other parallel processes using message channels. These basic features are included in the design of the basic processor, the Transputer device.

The SN1000 machine is a typical example of a transputer-based machine. It consists of a 48-node worker network that is built from three Supernode units [Supa], each consisting of 16 T800 workers, a T800 control Transputer and a 72x72 way RSRE
crossbar switch that provides interconnections between workers so that any N link can connect to any S link, and any W link can connect to any E link. The switch is fully programmable, providing the ability to dynamically reconfigure the network. All processors also have a control bus interface, which can be used for supervising the system, synchronisation and monitoring of applications. A Niche board is used to provide an interface to the Sun host. The T800-25 acts as the root processor for the network; one of its links is taken to a control Transputer, one to the T800-20, one connects into the worker network and the fourth to the SCSI disk server card.

Figure 2.1 presents an overview of the SN1000 machine in use at University College London.

The SN1000 machine can be used in conjunction with the 3L Parallel C & Tiny environment which controls both communication and synchronisation. This environment offers a complete software package that allows the development of parallel applications. Such applications are composed by a set of concurrent processes; the 3L tasks, that communicate with each other through their associated Tiny task, and the configuration specification that describes the actual location of each of the previously defined tasks.

2.3.1.2. SIMD Configurable: the Connection Machine

The Connection Machine (CM) is a Single Instruction Multiple Data (SIMD) machine configurable with up to 65,536 processors. Each processor has a single-bit processing unit and up to 256K bits of local memory, it is generally referred to as processor/memory cell. All the processors are controlled by a single instruction stream broadcast from a microcontroller.

Figure 2.2 shows a block diagram of the CM. Processors can communicate via a configurable communication network. The router operations allow any processor to write into the memory or read from the memory of any other processor. Additional system software provides other mechanisms for inter-processor communication, for example the scan operations which allows a quick summation of many values from
different processors into a single processors. In addition to this, the CM provides a floating point unit shared by every 32 processors, and a 32 bit number can be stored across 32 processors (one bit per processor). This is a way of sharing data among processors locally.

The CM uses a conventional computer such as a SUN-4 or VAX as a front-end machine. The original language for programming the CM is the Connection Machine Lisp (CmLisp) which corresponds to an extension of Common Lisp. It was designed to support the parallel operations of the CM, it is an abstract version of the CM. It mirrors the hardware of the CM, where the top level control is orchestrated by a conventional serial computer with thousands of values simultaneously calculated by the individual processor/memory cells.

2.3.1.3. SIMD Array: the Distributed Array of Processors

The DAP (standing for Distributed Array of Processors) is a Single Instruction Multiple Data (SIMD) machine which implements a fine grain data parallelism. Typically, data is spread across the memory of a large number of simple (1 or 8-bits) processors and the same operations is performed simultaneously on data objects in all or a selection of the processors. The processing elements (PE) within the DAP are organised in a square array, a 64x64 array in our case. Each PE has connections to its four nearest neighbour, referred to as north, east, south and west neighbours. Furthermore, fast data broadcasting and fetching is provided by a bus system that connects all the PEs in each row and all the PEs in each column. Each PE has its own local memory that ranges from 32 Kbits to 1 Mbits. A Master Control Unit controls the processor array by broadcasting instructions to each PE, which then operates on its own local data and this simultaneously with all other PEs. The PEs perform their basic operation; involving fetching or storing a memory bit within a single DAP cycle (of the order 100 nanoseconds). An overview of the DAP system is given in Figure 2.3.
The DAP is programmed using conventional languages that have been extended so to support parallel data constructs; enabling thus to benefit from the massive parallelism. Currently, the DAP can be programmed in Fortran Plus\(^2\) which corresponds to an extension of the Fortran language. This language allows algorithms to include parallel vector and matrices operations, and to consequently take advantage of their parallelism. Typically, a DAP application will be composed of a host program that handles the host-based I/O and user interaction, and a DAP program that uses the parallel features of the DAP. The DAP program resides in the code memory of the Master Control Unit, shown in Figure 2.8, and is accessible through a host computer (VAX or Sun). According to the application the control is wholly given to the DAP or the host remains in control; the DAP will then only run the highly data parallel functions.

### 2.3.2. Taxonomy

The three parallel machines presented above are representative of the spectrum of general-purpose parallel machines considered in this thesis, covering MIMD, SIMD configurable and processor array. As it has been done for neural systems, Table 2.2 summarises the features common to classes of general-purpose parallel machines. It also indicates the parameters which influence the performances of parallel machines. These parameters are intentionally kept at a high level. This is done because, the objective is firstly to analyse the parallelism exhibited by neural network systems, and secondly to map it onto parallel machines.

---

\(^2\) Fortran Plus is a trademark of AMT.
Table 2.2 - General Purpose Parallel Machines: Common Features & Examples -

A general-purpose parallel machine is composed of an architecture and a software environment. The architecture specifies the type of memory, the synchronisation and communication of the process. The software environment specifies the original programming language of the machine, and the tools available. These tools comprise tools dealing with inter-processes communication and tools dealing with translation/compilation of existing languages.

In Table 2.2, a general-purpose parallel machine is composed of an architecture and a software environment. The architecture is described by specifying the memory of the machine and the process synchronisation and communication. The memory includes the type, for example, distributed in the case of the SN1000 Transputer-based machine, and the size of the memory of each processor, referred to as $P \text{ Size}$ in the table. The process can generally be synchronised either via message-passing or via a central clock, and can use a network or a bus for communicating. The software environment describes the language initially devised for programming the machine, and also the tools available for dealing with inter-processor communication and use of other languages. As shown in the table, in the case of the Connection Machine, CmLisp (an extension of Common Lisp) was developed. Both the Connection Machine and the SN1000 machine provide tools for translating existing languages into the machine’s programming language.

2.4. Summary

This chapter has firstly shown the lack of consideration of neural networks’ parallelism (in programming them) and the lack of exploitation (in executing them). It has thus given material to declare the motivations for the work developed in this thesis, i.e. analysing and exploiting the parallel nature of neural systems. Secondly, it has presented typical examples of neural systems and parallel machines, as well as provided a taxonomy for each. The next chapter concentrates on the analysis of the parallelism of both neural systems and machines. Based on this analysis, it then identifies the features required by a mapping system for neural networks onto parallel machines.
Chapter 3
Parallelism in Neural Systems & Parallel Machines

This chapter analyses the parallelism in neural systems and machines. It identifies and classifies the parallelism embedded in neural systems. It then investigates the types of parallelism implemented by parallel machines and discusses the various approaches to parallel programming. Based on this, it justifies the directions taken in this thesis, and gives supporting evidence for a System for decomposing and mapping neural systems onto parallel machines.

3.1. Overview

As indicated in Chapter 1, the ultimate goal of this thesis is to develop a mechanism able to decompose and map virtually any neural system onto general-purpose parallel machines. Two tasks are involved, as shown in Figure 3.1. Firstly the expression of parallelism of neural systems. Secondly the exploitation of this parallelism across a range of parallel machines.

Figure 3.1 - Decomposing and Mapping Neural Systems onto Parallel Machines -
Decomposing and mapping neural network systems deals with two problems: the expression of the parallelism of neural systems NNSi, and its exploitation onto parallel machines M0.

In order to identify and justify the characteristics of the mechanism developed in this thesis, the parallelism of both neural systems and machines is examined. The parallelism of neural systems is classified into types. This classification is general in that it permits the identification of any form of parallelism implemented by any neural system. Parallel machines are seen as the support for the exploitation of neural systems' parallelism. Hence, the parallelism implemented by parallel machines is firstly presented. Secondly, parallelism as expressed by parallel languages is discussed. Thirdly, the various ways of dealing with the programming and exploitation of these
machines are investigated. Based on these analyses, the features of the mechanism for decomposing and mapping developed in this thesis are identified and the consequent choices are justified.

3.2. Parallelism in Neural Systems

Chapter 2 has provided a framework for the description of neural systems which defines them as formed by a neural model, a neural application, and their combination. This framework is now used to analyse the parallelism exhibited by neural systems, i.e. the types of parallelism found are identified relative to neural model and system processing.

3.2.1. Types of Parallelism in Neural Systems

The parallelism exhibited by neural systems can be divided into two distinct classes:

- model related,
- system related.

Firstly, the model related class concerns a neural model and its different processing phases. Secondly, the system related class concerns the actual processing cycles involved during the training of a neural system. Figure 3.2 shows the relations of dependence between types of parallelism and the components of a neural system.

![Figure 3.2 - Types of Parallelism versus Neural Systems' Components](image)

Figure 3.2 - Types of Parallelism versus Neural Systems' Components - A first type of parallelism is to be found in the processing performed by the neural system, referred to as system processing. The neural model itself exhibits various types of parallelism: within a neuron, within and between sets of neurons or connections.

The model related type of parallelism is exhibited when neurons operate concurrently, or when the network interconnectivity specifies groups of neurons or connections as computing in parallel. Thus, within model parallelism, three sub-types can be identified: i) parallelism within a neuron, ii) parallelism within a slab of neurons or connections, and iii) parallelism between slabs of neurons or connections. The notion of slab introduced here refers to a set of neurons or connections. This definition is
intentionally kept general in order to accommodate all types of neural models, and especially their alternative organisations. These three types of parallelism are further defined:

- **intra-neuron**: this type of parallelism is expressed within the individual rule that each neuron is performing. Typically, when a neuron computes its new state of activation, it has to compute its net input: \( net_i = \sum_j w_{ij} x_j \). In this function each individual multiplication is performed in parallel and the results are then summed. This type of parallelism and the associated decomposition is shown in Figure 3.3.

![Intra-neuron Parallelism Diagram](image)

\[ \text{Vectors} \]
\[ \begin{array}{c|c|c}
    w(1i) & x(1) & w(1n) \\
    \vdots & \vdots & \vdots \\
    \vdots & \vdots & \vdots \\
    w(ni) & x(n) & \text{Sum W.X} \\
\end{array} \]

\[ \text{Processors} \]

Figure 3.3 - Intra-neuron Parallelism -
To exploit the intra-neuron parallelism, the connection weights of a neuron \( w_{ij} \), and the state of the neurons each connection originates from \( x_j \) respectively form two vectors of \( n \) elements. In case of a processor array, these vectors can be directly mapped onto a column (or row) of processors. Each processor multiplies its weight and its state. The results are then summed.

- **intra-slab**: this type of parallelism is embedded in the fact that each neuron (or connection) generally performs its rule in parallel with all the neurons (or connections) of a same slab. For example, in the backpropagation model, all the weights which connect into the hidden layer can be updated in parallel.

- **inter-slab**: this type of parallelism is found at the highest level of a neural model, i.e. it addresses the global computations performed by a model. This global control specifies parallelism by stating that a particular slab \( A \) is to perform a rule \( X \) in parallel with a slab \( B \) performing a rule \( Y \). Still taking a backpropagation model as an example, it specifies that the weights which connect into the hidden layer and the weights which connect into the output layer can be updated in parallel.

The system related type of parallelism takes two aspects: one that defines parallelism over a set of input patterns, and one that defines parallelism over a set of net-modules in the model. The parallelism over a training set is generally exhibited by supervised systems. For example, in backpropagation systems, the training of an individual pattern can operate in parallel with other training patterns. In this case, the process of updating the weights is additive for the different training patterns, defining thus a parallelism over the training patterns. Figure 3.4 shows this type of parallelism
and the method for decomposing the neural system for exploiting it.

![Neural System](image)

**Figure 3.4 - Training Parallelism -**
The set of training patterns \( t(0), t(1), ..., t(p) \) of a neural system is split into \( x \) groups. \( x \) depends on the number of available processors on the parallel machine. In the figure \( x=2 \). The neural system is thus replicated 2 times, each replicated system operates on its particular set of patterns.

The parallelism over a set of net-module is generally exhibited by heterogeneous neural systems. Such neural systems are composed of modules that cooperate so as to achieve a higher function [Bot91a]. In this case, the neural application interacts with the neural model through different modules, and thus determines parallelism over the modules. This type of parallelism could be classified as model-related because it involves modules (as part of a model) that operate in parallel. However, the application and system processing are the components that actually determine this parallelism (the neural application interacts with the neural model through different modules). Thus, this justifies the decision to classify this type of parallelism as system-related rather than model-related. The types of parallelism are summarised in the table below, Table 3.1.

<table>
<thead>
<tr>
<th>Types of Parallelism</th>
<th>Neural System Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Related</td>
<td>training set</td>
</tr>
<tr>
<td></td>
<td>net-module set</td>
</tr>
<tr>
<td></td>
<td>system processing</td>
</tr>
<tr>
<td></td>
<td>system processing and application</td>
</tr>
<tr>
<td>Model related</td>
<td>intra-neuron</td>
</tr>
<tr>
<td></td>
<td>intra-slab</td>
</tr>
<tr>
<td></td>
<td>inter-slab</td>
</tr>
<tr>
<td></td>
<td>neurons (connections) local operations</td>
</tr>
<tr>
<td></td>
<td>organisation of the network (neurons and connections)</td>
</tr>
<tr>
<td></td>
<td>network dynamic processing</td>
</tr>
</tbody>
</table>

**Table 3.1 - Types of Parallelism in Neural Systems -**
There exists two major types of parallelism: system related, and model related. They address respectively the different levels of a neural system. System related parallelism consists of the training set type, and the net-module type. Model related parallelism consists of the intra-neuron type, the intra-slab, and the inter-slab.

An additional constraint to consider is the configuration of a neural system. Configuration is defined as the organisation and number of neurons and specific variations in the network interconnectivity that the model exhibits. The (non) existence
of particular types of parallelism is not dependent upon any actual configuration. That is, the parallelism is not directly affected by the configuration as such. Whatever configuration a model adopts, the parallelism remains conceptually the same. However, when mapping schemes are considered, and more particularly when they are evaluated, the configuration becomes a critical factor, and must be taken into account. These issues of determining and evaluating the mapping schemes of neural systems are considered in the next chapter. At this point, the spectrum of neural systems and their associated types of parallelism are considered.

3.2.2. Parallelism across Neural Systems

This section shows how the types of parallelism of neural systems are shared among the spectrum of neural systems. Neural systems have been classified into optimisation networks, supervised networks and unsupervised networks [Fal87a]. However, for our purpose they are classified according to the organisation of their neurons and connections (structured, unstructured) and their processing mechanisms (synchronous over the neurons or not). Indeed, it is the organisation of a neural system and its processing mechanisms that differentiates them with respect to parallelism.

The computations associated with neurons which are the basic elements of a neural system, mostly involve the computation of their net input. Indeed, the response of a neuron is governed by the state of the neurons it is connected with and by the weight of these connections. This is referred to as the neuron's net input and is typically of the form: \( \sum w_{ij} a_j \) where \( w_{ij} \) = weight for connection linking neuron \( j \) to neuron \( i \), and \( a_j \) = some status information about neuron \( j \). This indicates that the intra-neuron parallelism is characteristic of all neural systems.

The intra-slab parallelism is also present in almost all neural systems. Neural systems, either structured or unstructured, define slabs of neurons or connections that can operate in parallel. In the case of structured systems, there are layers of neurons or connections that perform the same rule and operate concurrently. This intra-slab type is represented by an intra-layer parallelism in the case of backpropagation model [Wer88a], or by an intra-cluster parallelism; in the case of competitive learning models [Rum87a]. In the case of unstructured systems; i.e. systems in which models are made of a unique large slab of neurons (and thus connections), neurons are performing in parallel with each other. However, the type of neural systems in which processing is carried out in an asynchronous fashion does not allow neurons to operate in parallel. For example, a variation of Hopfield system indicates that neurons update asynchronously with a probability \( p \) of being selected for updating, and therefore does not exhibit intra-slab parallelism.
Inter-slab parallelism is naturally found in structured neural systems. Layered neural networks, such as the backpropagation, are typical examples. However, when exploiting this inter-slab parallelism one might consider its potentiality (its benefit) against the one provided when exploiting the intra-slab or intra-neuron parallelism. Indeed, it might be more beneficial to exploit a lower grain parallelism; i.e. intra-neuron or intra-slab. These issues are considered in the next chapter.

System related parallelism is more difficult to identify when investigating a spectrum of neural systems. A main reason for this is that this type of parallelism has been mainly investigated only for backpropagation systems, and relates to the non-linearity of the transformations implemented by the neural system. The parallelism over the training input patterns implies that the weights adaptation for the training patterns is additive. The system can be trained in parallel for different patterns and the weight changes are then accumulated. These aspects have to be studied further before being generalised; however, it is reasonable to assume that this parallelism might be shared by other neural systems, and is thus worth considering.

The relationships between classes of neural systems and types of parallelism are summarised in Table 3.2.

<table>
<thead>
<tr>
<th>Types of Parallelism</th>
<th>Neural systems classes &amp; typical examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra-neuron</td>
<td>All systems e.g. Hopfield Backpropagation Competitive Learning</td>
</tr>
<tr>
<td>intra-slab (cluster) (layer)</td>
<td>Synchronous models e.g. Competitive Learning Learning Vector Quantisation</td>
</tr>
<tr>
<td>inter-slab (layer)</td>
<td>Structured models e.g. Backpropagation</td>
</tr>
<tr>
<td>training (batch)</td>
<td>Linear or quasi-linear systems e.g. Backpropagation</td>
</tr>
</tbody>
</table>

Table 3.2 - Neural Systems & Types of Parallelism - This table summarises how the types of parallelism are shared among the spectrum of neural systems. All neural systems exhibit intra-neuron parallelism, as opposed to the intra-slab parallelism which is present only in structured models, and similarly for the inter-slab parallelism. Finally it is the linear or quasi-linear neural systems that exhibit training parallelism.

3.2.3. Conclusion

This section has shown how the parallelism exhibited by neural systems can be classified into types. Furthermore, it has directly related these types to the framework for neural systems presented in Chapter 2. The types of parallelism relate to the neural model or the system processing, and are further refined to address their respective
elements. For example, the intra-neuron parallelism addresses the local operations performed by neurons. This has two consequences: firstly, the framework for describing neural systems can also be used to express their parallelism, and secondly the task of automatic parallelisation of neural systems is eased (compare with an automatic parallelisation of general-purpose tasks).

3.3. Parallelism in Machines

One of the key issues in this thesis is to fit the parallelism of our domain, neural systems, onto the machines' parallelism. In this respect, neural systems determine a parallel machine relative to the parallelism it exhibits/allows. Parallel machines have been examined in Chapter 2. They are now investigated relative to parallelism, firstly by describing the different types of parallelism implemented by machines, secondly by discussing how parallel languages reflect parallel machines, and thirdly by reviewing the approaches in programming and exploiting parallel machines. These studies will allow us to justify the directions taken in this thesis for solving the decomposition and mapping of neural systems.

3.3.1. Types of Parallelism

A useful classification of parallelism in machines is one based on types or level of parallelism exhibited. The set of machines considered in this thesis as general-purpose parallel machines are machines based on both synchronous architectures and MIMD architectures. From this set, four major types of parallelism can be identified: - process level, - block level, - instruction level, and - sub-instruction level. More specifically, they are defined as follows:

- **process level parallelism**: where processes are executed concurrently over different processors,
- **block level parallelism**: where instruction blocks are executed concurrently over different processors,
- **instruction level parallelism**: where instructions (that are in the same program segment) are executed concurrently over different processors (or on different execution units of the same processor),
- **sub-instruction level parallelism**: where an instruction is split into micro-operations which are then executed concurrently.

The process level parallelism corresponds to, for example, two independent high level tasks executing simultaneously, synchronising/communicating via dedicated channels. Considering the block level parallelism, an example is the concurrent execution of two different "for" loops. The instruction level parallelism occurs, for example within one of
these "for" loop in which instructions are executed concurrently. Finally, the machine level or sub-instruction parallelism is found where an instruction is decomposed into micro-instructions that are executed concurrently. These four types of parallelism can be found in the machines described earlier. MIMD type exhibits process level parallelism, SIMD type exhibits instruction level parallelism, Vector type exhibits block level parallelism, and systolic type exhibits sub-instruction level. This short classification is not, however, mutually exclusive.

Considering the size involved in the four levels of parallelism as a criteria, then process and block level parallelisms can be referred to as coarse grain parallelism, while instruction and sub-instruction parallelisms can be referred to as fine grain parallelism. Coarse grain parallelism is mainly exploited by distributed memory, loosely coupled multiprocessor architectures. In this case, the issue of interprocessor communication is a crucial factor for the whole parallel machine. This is referred to as the communication bottleneck where a processor cannot proceed at its maximum speed because it needs communication from another processor and the communication network is out of band. The key is to establish a trade off between the partitioning of a neural system in large amounts of small instruction blocks (for which communication bottleneck arises) and in small amount of bigger instruction blocks. Transputer-based machines, e.g. Meiko [Zha90b], Snode [Par89a], are typical examples of such parallel machines. The fine grain parallelism is generally exploited by global memory, tightly coupled multi single processor. In this case, the problems to solve are the parallelisation of memory access; i.e. this is referred to as memory contention. The Connection machine [Hil85a], Warp [Kun87a] are typical examples of such parallel machines. As we shall see later, this determination of levels of parallelism in machines can be related to the types of parallelism exhibited by neural systems. This will prove useful when quantifying the possible mapping schemes associated with neural systems.

3.3.2. Parallelism in Languages

It is often the case that a parallel language is developed for a given parallel machine. Such a language reflects thus the machine and its underlying architecture. This is shown by presenting two examples of parallel languages: Occam [INM84a] and Fortran Plus [AMT90a]. The usage of each language and its consequences on parallelism are illustrated by presenting an example of a matrix multiplication. Occam is described to represent coarse-grain parallelism and processing-based decomposition. Fortran Plus is described to represent fine-grain parallelism and data decomposition.
**Occam**

Occam, originating from Hoare’s Communicating Sequential Process (CSP) [Hoa78a], is based on processes that may be executed concurrently and which communicate using channels. A process, the fundamental working element of Occam, is a single statement, group of statements, or groups of processes. Programs in Occam are constructed from three primitive processes: assignment for example for example $x := e$, input for example $c ? x$ in which $c$ represents the value of a channel $c$, and output, for example $c / e$. A channel is an unbuffered structure that allows information to pass in one direction only, synchronising the transfer of information. Thus, a channel behaves as a read-only element for a receiving process and as a write-only element for a transmitting process. Three control constructs are provided: `SEQuential`, `PARallel`, and `ALTemative`, as well as the traditional `IF`, `FOR`, and `WHILE` constructs. The direct implementation of Occam is a network of processing elements, each which executes a process concurrently. A Transputer is such a processing element.

The matrix multiplication example, presented in Figure 3.5 outlines a systolic implementation running on a $n \times n$ square array of Transputers, each with four nearest neighbour connections. Figure 3.5 presents both the implementation on a Transputer grid and the associated Occam code.

![Processor Grid](image)

```
PROC mult (CHAN up, down, left, right)
VAR r, a, b;
SEQ
  r := 0;
SEQ i FOR n
  SEQ
    PAR
      up ? a
      left ? b
      r := R + (a*b)
    PAR
      down ! a
      right ! b
```

**Figure 3.5 - Matrix Multiplication on Transputer Grid**

The left hand side of the figure displays the grid of processing elements and the flow of processing and data. The right hand side of the figure displays the actual Occam code.

The left hand side of Figure 3.5 shows that the matrix A passes through the array in the horizontal direction, while the matrix B passes vertically. Each processing element accumulates an element of the resulting matrix R. Data must flow in the array so that data item $a[i,k]$ meets data item $b[k,j]$ in processing element $p[i,j]$. The right hand side of Figure 3.5 presents the associated Occam code.
The parallel algorithm, as described in Occam puts the emphasis on the processing elements. That is, the algorithm is devised by considering the processing each Transputer should perform. The data are finally located onto the Transputer grid based on this processing-based decomposition.

**Fortran Plus**

As stated in [AMT90a] Fortran Plus is intended to give a programmer a clear abstraction of the Distributed Array of Processors (DAP), and so its features map simply onto the DAP hardware. Fortran Plus is an extension of Fortran and has evolve to become a highly data parallel language of its own, only keeping some basic syntax and loop structure from Fortran. It is a vector and/or matrix based language in which parallelism is expressed by specifying parallel data vector and/or matrix. It provides syntactic extensions for the manipulation of these parallel objects. Firstly, matrices and/or vectors can be assigned or be part of an expression with no loop. For example, the assignment: \( M = I \) assigns each element of a parallel matrix \( M \) of \( n \) by \( n \) element to a same value \( (I) \). Secondly, Fortran Plus provides indexing techniques which allow operators to act on parts of matrices and/or vectors. For example, \( M(I,) = 2 \) assigns the value 2 to the first row of the matrix \( M \). Thirdly, it provides in-built functions performing special operations, for example \( SUMC(M) \) sums of the elements of each row of a matrix.

The matrix multiplication example, presented in Figure 3.6 outlines a standard implementation running on a \( n \) by \( n \) square array of processing elements. The figure presents both the implementation on the processor grid and the associated Fortran Plus code.

![Figure 3.6 - Matrix Multiplication on Processor Array](image)

*Figure 3.6 - Matrix Multiplication on Processor Array*

The left hand side of the figure displays the matrices of data and their location onto the grid of processing elements. The right hand side of the figure displays the actual Fortran Plus code.

The Fortran Plus code of a matrix multiplication, displayed in the right-hand side of Figure 3.6, is solely composed of a loop and one line describing operations on parallel
data matrices. This is quite different from the previous Occam code. As shown in Figure 3.6, the matrix $B$ simply maps onto the processor grid, whilst for each iteration in the loop, a row of matrix $A$ is replicated to form a new matrix. For each iteration, the vector $R(\cdot)$ is obtained by summing each column of the matrix obtained by multiplying $B$ with the replicated matrix $MATC(A(i, \cdot), n)$ (in this case, multiplication of two matrices refers to the multiplication of each individual element, i.e. $A(i, \cdot) \times B(i, \cdot)$).

The algorithm for the matrix multiplication is designed by primarily considering the matrices of data. It is the investigation of the possible arrangements of the matrices onto the processor grid that lead to the design of the algorithm.

There are two major ways for dealing with decompositions: processing-based as illustrated by Occam, and data-based, as illustrated by Fortran Plus. Processing-based corresponds to functional decomposition which divides up the function of a program and operates in parallel on the different parts. Data-based corresponds to domain-decomposition which divides up the data of a program and operates in parallel on the different parts. We now consider the various ways for dealing with the programming and exploitation of parallel machines.

3.3.3. Approaches for Programming & Exploiting Parallel Machines

A parallel machine and its associated environment provides a user with the capability of expressing and/or exploiting a program in a parallel manner. The expression of parallelism corresponds to the nature of programming parallelism. The exploitation of parallelism corresponds to the nature of executing parallelism. Taking expression and exploitation as classification criteria, three approaches are identified:

- **programmed**,  
- **automatic**,  
- **semi-automatic**.

The programmed approach provides both expression and exploitation of parallelism. It defines parallelism as controlled by a programmer, e.g. via languages. The automatic approach provides exploitation of parallelism; it automatically parallelises programs, e.g. via compilers. Finally, the semi-automatic approach provides some sort of expression of parallelism by extending sequential languages, and also provides automatic exploitation.

 Concerning *programmed* parallelism, the simplest approach is to write a new program for a specific machine. Parallelism is explicitly expressed using the primitives available on the target machine. One can take advantage of the specificities of the machine and tune the program accordingly. An example of this is the Occam language which is based on the concept of Communicating Sequential Process [Hoa78a] and reflects the architecture of Transputer-based machines. To some extent, programmed
parallelism is flourishing, in that each parallel machine comes with an associated programming environment which supports the exploitation of the machine's parallelism. Processor arrays provide ways of specifying vector and/or matrix parallel operations, as shown in the example of Fortran Plus. However, programmed parallelism relies heavily on the user; i.e. the user must specify not only the parallel aspects of its application but also must provide all the necessary associated elements (e.g. data exchanges). Furthermore, programmed parallelism is tied to a particular parallel machine or to a set of similar parallel machines. For example, a set of primitives has been defined which applies to shared memory machines [Boy87a], and another set which applies to message passing [Ath88a].

**Automatic** parallelism is a radically different approach; programmers write programs in their usual sequential languages and a compiler is used to detect parallelism and transform the program to a parallel program suitable for executing on a target machine. Applications of such program transformation techniques mostly use languages such as Fortran [Bar92a]. Implicitly, parallel high level languages, e.g. Pure Prolog [Kal90a], are in the same line of work in that they free the programmer from managing parallelism. The compiler, and possibly a run-time system, decide which available parallelism to exploit, and provide the necessary parallel management functions; i.e. description and synchronisation of the parallel actions. In the case of functional languages, it is the property of the basic operators of the language that enables parallelisation. Parallelism at a high level is extracted from these basic operators and their combinations [Cla88a].

In between these two approaches to parallelism, a **semi-automatic** approach can be identified. Firstly, parallel constructs can be added to existing sequential languages. An example of this is concurrent Pascal [Han78a], or concurrent C. In this case, although the compiler still manages the parallelism, the static analysis of the parallelism is made easier by the provision of these parallel constructs. The extent of the decisions made by the compiler, regarding what can or should be executed in parallel are reduced. Lastly, there are languages in which parallelism is expressed explicitly but in a machine-independent high level manner. Actor [Agh86a], Linda [Zen92a] are some examples of this approach. In this case, languages provide abstractions and constructs for expressing parallelism, the compiler then must support these abstractions efficiently.

This thesis's goal is to develop a **System** for decomposing and mapping neural systems onto general-purpose parallel machines. This **System** corresponds to the conceptual design of a software that automatically maps neural systems onto a range of parallel machines. In this respect, it falls into the category of automatic parallelism. This has been proven difficult for general-purpose tasks, but we are concentrating on neural systems for which parallelism has been clearly classified, thus this task of
automatic parallelisation is eased. Furthermore, we have shown that neural systems are lacking methods for expressing their parallel nature, thus a subsequent aim is to provide such a method. In this respect, this work falls into the category of programmed parallelism.

3.4. Mapping Neural Systems onto Parallel Machines

Based on the previous analysis of parallelism in machines and neural network systems, the features of the mapping mechanism developed in this thesis are identified and the consequent choices are justified. Firstly, the approaches to programming parallelism are reviewed in this context, and the decision to opt for a neural system specification with explicit support for parallelism is justified. Secondly, the issues concerning the optimal transformation of this specification into its ready-for-execution parallel equivalent are discussed.

3.4.1. Expression of Parallelism

Obviously, the simple approach to parallelism; i.e. the provision of a new parallel program is of little interest in our context. There is no genericity in such an approach. The same applies when considering compilers for existing sequential languages to parallel machines. Moreover, the extraction of parallelism from sequential code is often targeted to shared memory tightly/moderately coupled multi processors [San92a]. In addition, this approach does not tackle the lack of expression of neural systems' parallelism. Considering automatic compilers for implicitly high level parallel languages, not only are they difficult to construct but they do not allow the user to tune up their programs. Users are unable to describe the specific nature of parallelism in sufficient detail [Sab88a]. Furthermore, such languages do not seem to be adequate for specifying neural systems [And89a]. Finally, the feasibility of designing such compilers for a wide range of parallel machines; i.e. covering the spectrum from SIMD to MIMD machines, is yet to be proven. This is also true for sequential languages which provide parallel constructs; they are not sufficiently generic to cover a spectrum of parallel machines [San92a].

Hence, it is sensible to advocate the use of a specification for neural systems which expresses parallelism. However, quite often, languages supporting explicit parallelism are restricted to the parallelism found on the target machine. They are efficient, because inefficient constructs are completely avoided, and they express the underlying parallelism of the machine; they may contain hardware oriented features [Hoc84a]. Thus, and in order not to be limited to specific machines, we aim to obtain a language that expresses parallelism at a high level. This language should provide abstractions for parallelism which can then be efficiently transformed for exploitation onto a range of target parallel
machines.
This implies, firstly, the specification for neural systems to allow clear identification of the different types of parallelism. Secondly, for the specification to be efficiently translated onto a variety of parallel machines, it has to draw distinctions between key issues which might be addressed by different types of parallel machines. The issue for the mapping is to have all the information it requires instead of trying to deduce missing information. The specification must not be biased towards a particular parallel machine, which will result in facilitating a particular type of translation while making another type nearly impossible. For example, for the specification to be transformable onto MIMD architectures, it is necessary to distinguish between computation and communication. While, in the case of a translation onto SIMD architectures this issue is less crucial, but synchronisation points have to be marked. Chapter 2 has presented a framework which is sufficiently generic to accommodate the description of a range of neural systems. This framework can also accommodate the expression of the types of parallelism neural systems might exhibit, as shown in this chapter. Based on this, the specification for neural systems is intended generic; i.e. support the specification of most neural systems while also expressing their parallelism.

3.4.2. Exploitation of Parallelism

Concerning the exploitation of parallelism, a theoretical model of parallel computation could have been chosen to represent neural systems as parallel entities ready for execution. However, it is fair to state that there exists no universally accepted theoretical model upon which parallel machines can be based. There is no general consensus as to which of these is the best [Gib89a]. The P-RAM (Parallel Random-Access) model of parallel computation is one of the most used [May88a]. Even a combination of plain English and widely know parallel constructs is used as a method to describe designs of parallel algorithms [Akl89a]. However, these models obliterate hardware aspects; the P-RAM is somehow unrealistic and requires major restrictions when being concretised onto an existing parallel machine [Ski90a]. This can be illustrated by considering two examples of parallel machines: the Distributed Array of Processors (DAP) [Par88a] and the Parsys SN1000 [Par89a] machines. These machines are based on radically different paradigms. The SN1000 machine is a Transputer-based machine and is representative of the Multiple Instruction Multiple Data (MIMD) machines. Each processor is autonomous, with its own memory, and can only communicate via a message passing mechanism. The DAP machine is a typical Single Instruction Multiple Data (SIMD) machine, and consists of a large number of tightly coupled small processors. Each processor has no control on its own, but is controlled via a global control unit. Considering these two machines, there is no unified model capable
of describing them both accurately/completely. To address the deficiencies of models such as the P-RAM, researchers are developing new models; e.g. the CRAM model [Siv92a]. Valiant [Val90a] proposes a bridging model for parallel computation. This model, the BSP (Bulk Synchronous Parallel) model aims to be equivalent to the Von-Neuman model, but for parallel computation. However, it is not the purpose of this thesis to propose such a unified model of parallel computation. Rather, parallel machines are seen as the vehicle for the exploitation of neural systems' parallelism. Therefore, the use of a theoretical model of parallel computation to represent neural systems as ready for execution parallel entities is not suitable.

Therefore, the decision is made to provide mechanisms enabling the transformation from a neural system specification into a targeted parallel machine language. The neural system specification provides abstractions for parallelism; it is parallel architecture independent. The key idea is to develop a scheme that enables neural systems to be efficiently transformed for exploitation onto a range of target parallel machines.

3.5. Summary

This chapter has classified the parallelism of neural systems into types, and has integrated them within the framework developed in Chapter 2. It has also examined the approaches to programming parallelism in machines. Based on this, it has discussed the various approaches for dealing with the expression and exploitation of the parallelism of neural systems. The proposed solution, summarised in Figure 3.7, firstly analyses a neural system and its parallelism, and secondly examines the targeted parallel machine in order to produce the mapping of the neural system.

Based on the investigation of the parallelism of both neural network systems and parallel machines, it has been decided to develop a mechanism able to identify the possible decompositions of neural systems (from a specification which explicitly supports parallelism) and able to adapt to various parallel machines to select the most suitable decomposition for the targeted machine. This mechanism is referred to as System for decomposing and mapping, and is developed in the next chapter.
The Neural System

Analysis & Extraction of the Parallelism

Decompositions

Programming a Parallel Machine

Mapping the Neural System onto the Parallel Machine

The Parallel Machine

Figure 3.7 - Expressing & Exploiting the Parallelism of Neural Systems -
The targeted neural system is analysed along with its parallelism, explicitly expressed. The targeted parallel machine is then examined along with the possible decompositions of the neural system in order to produce the most suitable mapping of the neural system onto the parallel machine.
Chapter 4

The Mapping System

This chapter describes the Mapping System for decomposing and mapping neural network systems onto general-purpose parallel machines which enables an automatic and optimal mapping. The System divides the mapping task into two sub-tasks: a neural system Abstract Decomposition module (AD) and a Machine Dependent Decomposition module (MDD).

4.1. Overview

The System for decomposing and mapping expresses the parallelism exhibited by neural systems, and exploits it across a range a general-purpose parallel machines. In doing so, the System is believed to have the following characteristics:

- generic - able to cope with virtually any neural system or parallel machine,
- transparent - the mapping scheme of neural systems are automatically produced,
- optimal - the mapping scheme produced is the most suitable for the neural system and machine,

The System provides mechanisms for the automatic identification and specification of the most suitable mapping scheme of a neural system. More specifically, it considers a neural system, extracts its parallelism, provides alternative decomposition schemes, decides which decomposition is most suitable, and finally generates the appropriate specification for the targeted parallel machine. The issues addressed are the identification of alternative decomposition schemes of neural systems, and the identification of an optimal mapping scheme. It is necessary, at this stage to distinguish between decomposition scheme and mapping scheme of neural systems, and also to specify what is meant by most suitable mapping scheme.

Terminology

A decomposition scheme is machine-independent and specifies how a neural system is decomposed onto an ideal parallel machine. In this case, the ideal parallel machine is defined as a network of processors with no specification concerning its management (e.g. communication schemes). Hence, in our context, a decomposition scheme defines the decomposition of a neural system in a machine-independent manner. It specifies the neural system’s elements that are distributed and this irrespective of machines’ considerations.
A mapping scheme of a neural system determines a decomposition scheme relative to a specific parallel machine. In this case, the parallel machine’s features are known, i.e. its type (SIMD, MIMD, Hybrid), its communication schemes, its number of processors, its type of processors (e.g. memory capacity).

A most suitable mapping scheme corresponds ideally to the optimal mapping scheme of a neural system onto a parallel machine. The mapping scheme selected aims to allows the maximum execution speedup $\Theta_{y,k}$. Formally, the optimal mapping scheme $M_y$ onto a parallel machine $PM_k$ is such that:

$$\Theta_{y,k} = \text{MAX} \left( \Theta_{x,k}, \Theta_{y,k}, \Theta_{z,k} \right)$$

where

$\Theta_{x,k}$ - the speedup factor for the mapping scheme $M_x$ onto the parallel machine $PM_k$ calculated such that $\Theta_{x,k} = T_s / T_p(x; k)$

$T_s$ - the execution time of a neural system on a sequential machine,

$T_p(x; k)$ - the execution time for the mapping scheme $M_x$ of a neural system onto a parallel machine $PM_k$.

These expressions state that, from the set of mapping schemes $M_x$ of a neural system, we aim to retrieve the mapping scheme $M_y$ on the parallel machine $PM_k$ with a speedup $\Theta_{y,k}$ verifying equation (4.1).

Decomposing & Mapping Neural Systems

The identification of the decomposition schemes of neural systems requires the extraction and analysis of their parallelism. This is machine independent in that only neural systems and their parallelism need to be considered. The characteristics of the machine intervene when tackling the problem of identifying and specifying the most suitable mapping scheme. To identify the optimal scheme, alternative schemes are assessed according to the targeted parallel machine. Based on this, the System distinguishes two phases for the mapping: a machine independent phase and a machine dependent phase. The machine independent phase is referred to as the Abstract Decomposition module, and deals with neural systems, their parallelism, and decomposition schemes. The machine dependent phase is referred to as the Machine Dependent Decomposition module, and deals with neural systems mapping schemes, their evaluation and the parallel specification of the selected one. It shall be noted that
the evaluation of mapping schemes presented in this chapter is informal, i.e., the System does not provide, in a strict sense the optimal mapping scheme as described in (4.1). Figure 4.1 gives an overview of the proposed System.

![Diagram](image)

**Figure 4.1 - The System for Mapping Neural Systems**  
Via the Abstract Decomposition module, the System for decomposing and mapping devises the alternative decomposition schemes of a neural system. Then, the Machine Dependent Decomposition considers the various decompositions relative to the targeted parallel machine, and generates the specification of the most suitable mapping scheme to implement.

As shown in Figure 4.1, the System is based on the division of the mapping task into a machine independent task, i.e. the Abstract Decomposition (AD), and a machine dependent task, i.e. the Machine Dependent Decomposition (MDD) of neural systems. More specifically, the role of the AD and MDD within the System is:

- **Abstract Decomposition** -  
The AD is responsible for extracting the parallelism that neural systems exhibit and for identifying the associated alternative decomposition schemes.

- **Machine Dependent Decomposition** -  
The MDD considers relevant machine criteria (e.g. machine size, communication mechanisms) and evaluates the alternative mapping schemes in order to recommend the most suitable one. It then generates the specification of the optimal mapping scheme for the targeted neural system and machine.

The AD module requires a specification for neural systems which encompasses abstract parallel features. The AD then analyses the neural system and determines the set of alternative decomposition schemes. Then, the MDD examines the decomposition schemes along with the targeted parallel machine. In other words, it defines the decomposition schemes according to the characteristics of the machine, i.e. it specifies the mapping schemes. The most suitable scheme is then translated by the MDD into its parallel equivalent onto the targeted machine.
This section has overviewed the System for decomposing and mapping neural systems onto general-purpose parallel machines this thesis proposes. The next two sections detail the two modules of the System, namely the Abstract Decomposition module and the Machine Dependent Decomposition module.

4.2. Abstract Decomposition

The Abstract Decomposition module analyses the parallelism inherent in a neural system and identifies its various decomposition schemes. This is done irrespective of machine considerations. The need for a specification for neural systems with parallel features of parallelism has been demonstrated in Chapter 3. Hence, this section firstly describes the nC neural network specification, which has been developed within the Pygmalion project [Tre89c]. Secondly, this section describes the alternative decomposition schemes of neural systems. The classification of the parallelism of neural systems into types forms the basis for the identification of decomposition schemes of neural systems. Each type of parallelism determines a decomposition scheme by itself. They are referred to as basic decomposition schemes. These schemes exploit one type of parallelism. However, they are not mutually exclusive; i.e. it is possible to exploit within a same decomposition scheme multiple types of parallelism. These schemes are referred to as multiple decomposition schemes.

4.2.1. nC: The Pygmalion Specification for Neural Network Systems

The nC specification was built as a target language to accommodate the requirements of neural network programming environments [Aze90a]. The nC data structure used to specify neural systems is firstly described. Then, using this generic data structure, the various stages required to construct a complete neural system in nC are presented. A complete manual for nC can be found in [Vel90a].

4.2.1.1. nC: the Data Structure

A neural system described in nC is represented by a hierarchical C data structure that embodies data and processing. This hierarchy defines a neural system as composed of networks that have layers, that are composed of clusters, that comprise neurons, that contain synapses. At each level of the hierarchy the operations performed, referred to as rules are specified. The following figure, Figure 4.2 presents the generic nC data structure, and also the declaration of the C structure representing a neuron,
typedef struct net {
    typedef struct layers {
        typedef struct cluster {
            typedef struct neuron_type{
                ...
            }
            ...
        }
        ...
    }
    ...
}

typedef struct NEURON neuron_type,
struct NEURON {
    int n_rules,
    rule_type *rules,
    int n_parameters,
    para_type *parameters,
    para_type state[N_STATES],
    int fanin,
    int fanout,
    struct NEURON **input_neuron,
    struct NEURON * *output_neuron,
    int synapses,
    synapse_type *synapse,
};

Figure 4.2 - The nC Data Structure & Declaration of the Neuron C Data Structure -
The top half of the figure displays the nC data structure. This data structure describes a neural network as a net composed of layers, which in turn are composed of clusters, themselves composed of neurons, composed finally of synapses. Each level in the structure is based on the same principles, i.e., each level comprises rule and data information as well as pointers to its sub structures. The bottom of the figure presents the neuron level. The description of rule and data consist of their number (e.g. n_parameters, n_rules) and their specification (e.g. para_type, rule_type). The neuron data structure also contains pointers to neurons it connects to.

4.2.1.2. Building a Neural System in nC

Using the generic nC data structure, a neural system is constructed by specifying the following four components:

1. the rule bodies - define the processing performed by each class of rules,
2. the connect procedure - transform the generic nC data structure into a specific one (according to the concerned neural system),
(3) the building procedure - construct the rules and assign them to the appropriate level in the nC data structure,

(4) the nC image creation procedure - apply the connect procedure and the building procedure to produce the complete nC memory image of a neural system.

Each of these components is now detailed.

(1) rule bodies

The rule bodies define the code of the various classes of rules performed by a neural system. Rules are organised into classes; a class determines the processing performed by a set of similar elements in the nC data structure. For example, a weight update rule class describes the operations performed by a set of connections in a generic manner.

These operations apply to a generic list of data parameters $p$. This list of parameters $p$ is instantiated once the building procedure is applied. Figure 4.3 describes the rule body of a rule that computes the new state of a neuron, according to a standard state update operation such that $x_i = \sum_j w_{ij} x_j$. This rule class can be applied to a set of neurons.

```c
/* state_upd_class */
int State_update (p)
para_type **p; /* parameter list p: */
    *p[0] = *state /*acc
{
    /* the value of the first parameter is assigned to the dot product of the */
    list of parameters starting at p[4] */
    p[1]->value.f = dp(p + 2);
    p[0]->value.f = 1.0 / (1.0 + exp(- p[1]->value.f));
    return(0);
}
```

Figure 4.3 - Rule Body of a Standard Neuron State Update -
The top half of the figure displays the data structure representing a rule, whilst the bottom half gives the actual C code of the State_update rule. The rule body specifies the processing performed by the rule in a generic manner. The operations performed apply to a generic list of parameters $p$.

(2) connect

The connect procedure converts the general data structure into a specific memory structure. It allocates memory for the rules and for their parameters, and also allocates synapses for the neurons. Moreover, it connects the neurons; i.e. it specifies the actual pattern of connectivity. The neuron's connections are described within the structure of a neuron. The neuron structure contains pointers to its incoming neurons; i.e., the neurons that connect into. These pointers are set by this procedure.

(3) building

The building procedure is the most crucial operation. It constructs all the rules, attributes function pointers, and assigns the rule parameter list. This is performed via the definition of the class of the rule, via its initialisation and via its extension.
The class of a rule specifies the number of initialisation and extension parameters to expect. It also specifies if the rule can operate in parallel, i.e. \textit{pexec} or sequentially, i.e. \textit{sexec}. Figure 4.2 shows two typical examples of rule classes.

\begin{verbatim}
class_type state_upd_class = { State_update,"State_update",2,2 }; class_type err_cal_meta_class = { pexec,"pexec",0,1 };  
\end{verbatim}

\textit{Figure 4.4 - Rule Class: Examples -}
The \textit{state_upd_class} specifies a rule type with two initialisation and extensions parameters, and also its reference to the rule body. The \textit{err_cal_meta_class} specifies a rule type that can execute in parallel, with no initialisation parameters and one extension parameter.

The \textit{initialisation} of a rule corresponds to the definition of the rule’s name, the rule’s class, the associated rule body, and the definition of the initialisation parameters the rule applied to. These parameters can be either data parameters (e.g. the state of a neuron) or another rule, in which case the rule decomposed into sub-rules. In the case of the neuron state update rule, which body is described in Figure 4.5, the initialisation is as follows:

\begin{verbatim}
rule_init ("neuron.std_upd", /* rule's name */
&sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[ NEU_R_state_upd ], /* rule addressed */
&state_upd_class, & /* rule's class */
/* these 3 parameters for the rule relates to the rule body p[1]->p[3] */
&sys->net[cn]->layer[i]->cluster[j]->neuron[k]->fanin,
&sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[ N_STATE ],
&sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[ N_ACC ],
EOP );
\end{verbatim}

\textit{Figure 4.5 - The Initialisation of a Data Rule -}
The rule "neuron.std upd" is assigned to a neuron \(k\) of a cluster \(j\) of a layer \(i\) of a net \(cn\). The rule is initialised by setting its name, its class (which links the rule to its body) and by instantiating its basic parameter list (as defined by the rule body).

The \textit{extension} of a rule corresponds to the instantiation of the extension parameters applied to the rule. For a rule that applies to data parameters it assigns each parameter to the data concerned. This is presented in Figure 4.6(a), which shows the extension of the state update rule. For a rule that applies to other rules, it assigns each parameter to the
sub-rule concerned, as illustrated in Figure 4.6(b).

/* reference to the rule */
rule_extend ( &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->rules[NEU_R_state_upd],
k,   /* current extension iteration */
/* these 2 parameters for extensions relates to the rule body p[4]->p[4+size] */
&amp;sys->net[cn]->layer[i]->cluster[0]->neuron[j]->synapse[k]->weight,
&amp;sys->net[cn]->layer[i]->cluster[0]->neuron[j]->input_neuron[k]->state[N_STATE],
EOP );

Figure 4.6(a) - The Extension of a Data Rule -
The state update rule of a neuron \( j \) is extended \( n \) times. The \( k^{th} \) extension assigns the weight of the synapse \( k \) to its first extension parameter, and assigns the state of the input neuron \( k \) to its second parameter.

An example of the extension of a rule which applies to other rules is now given. In this case, the extension parameters refer to rules rather than data parameters.

/* reference to the rule */
rule_extend ( &sys->net[cn]->layer[i]->rules[LAY_R_state_upd],
j,   /* current extension iteration */
/* the parameter for extensions; i.e. another rule */
&amp;sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_state_upd],
EOP );

Figure 4.6(b) - The Extension of a Rule Applying to other Rules -
The layer state update rule is extended \( n \) times. The \( j^{th} \) extension assigns its parameter to a pointer to the state update rule of a cluster \( j \).

It is the information contained in the initialisation and extension of rules that are crucial for the any mapping system in order to be able to translate the data structure into its parallel equivalent. The class of each rule indicates its possible parallel execution, and its parameters indicate which data the rule applies to. However, there is a lack of information in this initial nC structure in terms of data requirements, especially for translation to MIMD-based machines. These issues are considered in detail in the next chapter.
The creation procedure simply constructs the nC image of a neural system by applying the connect and building procedures to the generic nC data structure. The application of these two procedures allocates the memory and sets the data structure which represents a specific neural system. The complete nC memory image of a neural system is thus obtained.

4.2.2. Neural System Decomposition Schemes

A neural system is described as an nC structure which expresses both data and processing. The types of parallelism exhibited by the neural system are made explicit by the structure. Decomposition schemes are determined according to the type of parallelism they exploit. Therefore, the extraction of the types of parallelism and associated data and processing information allow the identification of the alternative decomposition schemes of a neural system. The purpose of this section, as part of the description of the neural system Abstract Decomposition module, is to show how neural systems decomposition schemes are identified. Issues concerned with their automatic identification are addressed later on.

In machine independent terms, a decomposition scheme specifies how an application is decomposed onto a virtual parallel machine. In this case, a virtual parallel machine represents an ideal machine which is bottleneck-free and constraint-free. The machine is seen as a network of processors with no description concerning its size or its management; e.g. communicating processors, central clock... In our case, a decomposition scheme defines the partitioning of a neural system into parallel/concurrent\(^3\) components. Decomposition schemes relate type of parallelism and their exploitation to the considered neural system. They define the elements and the computational phases of a neural system involved in the decomposition, and consequently specify which type of parallelism is exploited.

Decomposition schemes incorporate two types of information: \textit{spatial} & \textit{temporal} information. Spatial information will determine the neural system data layout. Temporal information define the series of sequential-parallel computations. They are not machine dependent and can be described in an abstract manner. Temporal information could be represented as a task-precedence graph [Sev89a]. Spatial information does not specify which particular data is to be located on which particular processor, but defines the abstract data decomposition in which a set of data items is distributed. The description of the spatial information of a decomposition scheme is equivalent to a specific domain

\(^3\) As already stated we do not emphasise the difference between concurrent and parallel.
decomposition. The specification of the temporal information of a decomposition scheme is equivalent to a functional decomposition. Domain decomposition, and functional decomposition are known techniques for parallel programming [Cha92a]. Domain decomposition is used in the context of large data domain problems, while functional decomposition is used for problems for which the major aspect is the function, the algorithm. In our case; i.e. decomposition schemes for neural systems, the emphasis is on neither of these techniques. Rather, decomposition schemes are firstly identified on the basis of the type of parallelism they exploit. They are then described in terms of both spatial and temporal information (using the framework and terminology presented earlier).

There exist basic decomposition schemes which exploit one type of parallelism, and multiple decomposition schemes which exploit multiple types of parallelism. The basic decomposition schemes are firstly considered. Secondly, the multiple decomposition schemes are discussed.

4.2.2.1. Basic Decomposition Schemes

Neural systems exhibit model-related parallelism and/or system-related parallelism. Based on this, two decomposition schemes are identified:

- model based decomposition - In this case, the neural model is decomposed along its elements; connections for example.
- system based decomposition - In this case, the neural system is decomposed along either its input patterns or its net-modules.

These decomposition schemes can be further described according to the specific type of parallelism they exploit. This is done in Table 4.1 which presents the spatial and temporal information of these schemes and relates them to the type of parallelism they exploit.

---

4 This refers to level, phases etc., presented in chapter 2 and refined in chapter 3 to include information about parallelism.
Type of Parallelism exploited | Data Decomposition (spatial information) | Function Involved (temporal information) | Decomposition Scheme
--- | --- | --- | ---
model related | intra-neuron | set of connections | weighted input neuron operation | model based connection
intra-slab | inter-neurons | set of neurons | neuron operation | neuron connection
inter-connections | inter-slab | set of slabs | slab operation | neuron-slab connection
inter-neuron-slab | inter-connection-slab | set of neuron-slabs | neuron-slab operation | connection-slab
inter-connection-slab | system related | net replication | system operation | net replication
net-module | training (batch) | net-module replication | net-module operation | net-module replication

Table 4.1 - Spatial & Temporal Information Defining Decomposition Schemes

The exploitation of a type of parallelism determines particular data decomposition and parallelised function (function which addresses different levels of a neural system). The table indicates the two major types of parallelism; model and system related. It details them and associates a specific basic decomposition scheme to each of them. For example, the exploitation of intra-neuron type of parallelism involves a decomposition along the connections, the function parallelised is the weighted neuron computation. This scheme is referred to as 'connection' in the Table.

From Table 4.1, six basic decomposition schemes are defined: connection, neuron, neuron-slab, connection-slab, net replication, and net-module replication. Each of these schemes is now detailed.

The system related parallelism includes the training parallelism and the net-module parallelism. The training parallelism is exploited via the net replication decomposition scheme (as shown in Table 4.1) which replicates the whole system across a network of processors. In this case, each replicated system acts upon a particular training pattern. The temporal information indicates that it is the highest neural computation that occurs in parallel; i.e. the system level processing in Table 4.1. In terms of spatial information, this decomposition scheme implies the replication of the entire neural net, as depicted in Figure 4.9. Similarly, the net-module parallelism is exploited by replicating the various net-modules across a network of processors; referred to as the net-module replication scheme. This case concerns neural systems formed by two or more net-modules, in which each net-module is in fact performing a high-level transform of its own. For example, one can imagine a system composed of a net-module implementing a self-organising map and a second net-module implementing a backpropagation. It is these high-level transforms that will occur in parallel (temporal information), by replicating the net-modules (spatial information).
Within the model related parallelism, there exist several alternative sub-types of parallelism: intra-neuron, intra-slab, and inter-slab.

The intra-neuron parallelism is exploited if each individual multiplication (connection weight by input state) can occur in parallel. It should be noted that one might specify a neural system that does not perform such multiplication. However, neurons react according to their environment, which is 'encoded' via their weighted connections. Hence, it is sensible to claim that any neural system operates some sort of transform on input neurons parameters by the connections weight. In terms of spatial decomposition, the intra-neuron parallelism is achieved if the connections are distributed over the processors; this is referred to as connection decomposition in Table 4.1. This scheme is pictured in Figure 4.7. In this case, the function parallelised relates to the computation occurring within a neuron, i.e. part of the neuron computation occurs in parallel.

The intra-slab parallelism refers to the parallelism occurring within a slab of neurons or connections. To exploit this type of parallelism, neurons which belong to one slab are spatially distributed over the processors; this is referred to as the neuron decomposition. As indicated in Table 4.1, it is the basic operations performed by a neuron which are parallelised. As opposed to the previous one (where it is the computations occurring within a neuron which are parallelised) it is the overall neuron transform which is parallelised, i.e. the neurons themselves compute in parallel. Similarly, to exploit the inter-connections parallelism, the connections are distributed over the processors. This spatial decomposition is similar to the one determined when exploiting the intra-neuron parallelism. It is referred to as connection decomposition (Table 4.1). In this case, only a certain group/slab of connections are distributed, as depicted in Figure 4.7 for two slabs of connections.

Finally, to exploit the inter-slab type, the slabs of a neural model are decomposed. This decomposition is referred to as slab decomposition (Table 4.1), where a slab can refer to layer or cluster. It is the functions of the slab which are parallelised; i.e. temporal information in the table. When a slab represents a group of connections, the decomposition is referred to as connection-slab decomposition, whilst it is referred to as a neuron-slab decomposition when slab represents a group of neurons.

The following figures, Figure 4.7 to 4.9, represent pictorially the various basic decomposition schemes.
Figure 4.7 - Connection Decomposition Scheme -
Connection decomposition distributes the connections of a single slab over a network of processors, marked \( w^1 \) for the first layer, and \( w^2 \) for the second layer. Each ideal processor embeds a connection of each slab and its associated computation.

Figure 4.8 - Neuron Decomposition Scheme -
Neuron decomposition distributes the neurons of a single slab over a network of processors, marked \( N^1, N^2, N^3 \). Each ideal processor embeds a neuron of each slab and its associated computation.

Figure 4.9 - Net Replication Scheme -
Net replication simply replicates the entire neural system, marked NS, over a network of processors. Each ideal processor embeds the neural system and its computations; the high-level transform in this case.

Basic decomposition schemes can be combined to form multiple decomposition schemes which exploit more than one type of parallelism. They are now presented.
4.2.2.2. Multiple Decomposition Schemes

Six basic decomposition schemes have been identified: the connection scheme, the neuron scheme, the neuron-slab scheme, the connection-slab scheme, the net replication scheme, and the net-module replication scheme. These can be combined in groups of two, three, four, five or six which gives a total number of decomposition schemes of \( C_2^2 + C_3^2 + C_4^2 + C_5^2 + C_6^2 \). These schemes can be defined as direct combination of basic decomposition schemes. By doing so, the data distributed by a multiple scheme correspond to a simple integration of the data decomposition of the basic schemes they are composed of. And similarly for the functions involved in the parallelisation. Hence, it is not necessary to list the possible combinations exhaustively. The description of an example is sufficient; the same method can be followed to specify any other combination.

The example considers the multiple decomposition scheme formed by the combination of the neuron decomposition scheme and the net replication scheme. This scheme exploits both the intra-slab and training types of parallelism. It determines a spatial decomposition where the neural system is replicated over a network of processors, and where each replicated system distributes the neurons belonging to a same slab.

![Diagram of Multiple Decomposition Schemes](image)

*Figure 4.10 - Combination of Neuron Decomposition & Net Replication Schemes - The neural system is replicated over the network of processors, then each replicated system implements a neuron decomposition scheme which distributes neurons of a same slab across processors. In this case, there are more than one processor associated with each replicated system.*

As shown in Figure 4.10, the neural system is firstly replicated over the processors, as indicated by the net replication scheme. It is then decomposed along the neurons, as indicated by the neuron decomposition scheme. Similarly for the functions involved in the parallelisation: the neuron’s computations are parallelised as well as the high-level transform applied by the whole system.

Neural systems decomposition schemes have been identified relative to the types of parallelism they exploit. These decomposition schemes are now related to a targeted parallel machine, thus defining the mapping schemes associated with neural systems.
4.3. Machine Dependent Decomposition

The neural Machine Dependent Decomposition module performs two operations:

- evaluates the benefit of alternative mapping schemes of a neural system,
- provides the specification of the most suitable mapping scheme of a neural system.

These operations depend obviously on both the neural system and the targeted parallel machine; they can only be specified when the machine is known. Thus, the following sections describe the MDD module at an informal and conceptual level, i.e., neither the neural system nor the targeted parallel machine are defined. Firstly, the operations of the MDD are described, and its integration within the System is presented. Secondly, the alternative mapping schemes of neural systems are specified, this is done in generic machine-dependent terms. The process of selecting the most suitable mapping scheme is also given.

4.3.1. Overview

The Machine Dependent Decomposition module acts on a specific neural system and on a specific parallel machine. Figure 4.11 represents the MDD as part of the System for decomposing and mapping.

![Diagram of MDD Module: Selection & Translation](image)

*Figure 4.11 - The MDD Module: Selection & Translation - The MDD analyses the decomposition schemes of a neural system along with the relevant machine criteria. The neural system and its decomposition schemes are provided by the Abstract Decomposition module. The MDD integrates them with the targeted parallel machine to decide upon the most suitable mapping scheme to implement. It then generates its specification onto the parallel machine.*

As shown in Figure 4.11, the MDD is responsible for firstly determining the most suitable mapping scheme of a neural system, and secondly generating its parallel specification on the targeted machine. In Chapter 2, general purpose parallel machines were described as composed of an architecture and a software environment (cf Table
2.3), this was further refined in Chapter 3 to include parallelism. The translation process accomplished by the MDD obviously takes into account the software environment, in order to fully exploit the parallelism as expressed by the language, and also to benefit from additional tool available. Furthermore, the translation from the specification for neural systems to its equivalent on a parallel machine is made possible by the explicit parallelism supported by the specification itself. In addition, the construction of a prototype in the next chapter will validate pragmatically this translation operation. Therefore, this section puts the emphasis on the definition of neural system mapping schemes and on the selection process of the most suitable one.

4.3.2. Neural System Mapping Schemes

This section examines the effect that parallel machines' features have on the alternative decomposition schemes of neural systems, as defined by the Abstract Decomposition module. In doing so, the mapping schemes are determined and their suitability is assessed according to machines criteria. This results in the definition of a selection process for identifying the most suitable mapping scheme for implementing a neural system onto a parallel machine. Furthermore, as we shall see later (in Chapter 6), it will be possible to quantify the benefit of alternative mapping schemes, and to retrieve analytically the optimal one.

4.3.2.1. From Decomposition Schemes to Mapping Schemes

The Abstract Decomposition module has identified two types of decomposition schemes: basic and multiple. This classification is based on the type of neural systems' parallelism exploited. This section surveys these decomposition schemes and examines the effect of parallel machines relative to them, i.e., it specifies and compares the mapping schemes. This survey is done on the basis of the analysis of parallel machines' features carried out in Chapter 2 and 3.

The first and obvious feature of parallel machines to consider is the type of parallelism implemented by the targeted machine. Indeed, a machine which exhibits process level parallelism, for example, is well suited for implementing a net replication decomposition (a basic scheme where a neural system is replicated over a network of processors), but not suited for a connection decomposition scheme. It is also necessary to consider other features, such as the topology of the machine, or the processors memory, in order to evaluate the suitability of basic and multiple mapping schemes, and compare them with each other. At this point, it is important to note that the classification of decomposition schemes into basic and multiple schemes also applies to mapping schemes. Furthermore, similar naming conventions are applied, i.e., each named decomposition scheme has a corresponding mapping scheme. The next paragraph
examines the basic mapping schemes and analyses the required features of the parallel machines. Then, the study is extended to integrate multiple basic schemes. This is possible because multiple decomposition schemes have been defined as direct combination of single ones.

To determine the suitability of the basic mapping schemes, the machine features required to implement both the data and function decomposition (as defined by the Abstract Decomposition module in Table 4.1) are analysed. The connection mapping scheme requires the machine to implement instruction and sub-instruction levels of parallelism. This scheme distributes the connections of a neural system and parallelises the connection processing (instruction level) and the weighted input neuron operation (sub-instruction level). These levels of parallelism are typical of SIMD and systolic parallel machines. The processors of such machines are generally suitable; i.e. their memory and their processing power can handle the connections and their operations (the processing involves standard addition/multiplication). Concerning the neuron mapping scheme, the function parallelised; i.e., the neuron operations, might involved non-linear computation (e.g. exponential), in this case the processors have to allow such operations. The type of parallelism implemented by the machine and the communication mechanisms available are used to compare the connection mapping and the neuron mapping. The size of the neural system to map is also to be taken into account. This favours the connection mapping because the number of connections is usually larger than the number of neurons. This is true in the case of an SIMD type machine, however, in the case of an MIMD-based machine for which communication is one of the key problem, the neuron mapping is more suitable, i.e., it decreases the number of data exchanges. The neuron-slab and connection-slab mapping schemes require the machine to implement a high level of parallelism; block or even process level parallelism. This is also shown when examining the data decomposition involved which requires the processor memory to be able to contain a set of neurons or connections. This seems to indicate that such mapping schemes will be suited for MIMD-based types of machines. However, an SIMD machine can be adapted to implement such schemes. In this case, the network of processors is grouped to form a collection or "block" of processors of bigger size and capability. In addition to this, the communication mechanisms have to be examined in order to assess if the machine can efficiently accommodate such schemes. For example, a 1-dimension massively parallel machine will not be suitable. Lastly, it is the net or net-module replication mapping schemes which requires the larger type of processors. In this case, each processor needs to be able to handle a complete neural network (or module), in order to replicate the neural network (or module). MIMD-based machines which exhibit a coarse grain of parallelism seem to be the most suitable for implementing these schemes.
The following table, Table 4.2, summarises the characteristics of parallel machines required by each basic mapping scheme.

<table>
<thead>
<tr>
<th>Mapping Schemes</th>
<th>Machines Features</th>
<th>Topology, Communication, Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection mapping</td>
<td>Level of parallelism</td>
<td>1D or 2D topology</td>
</tr>
<tr>
<td>Neuron mapping</td>
<td>instruction &amp; sub-instruction</td>
<td>1D or 2D topology communication mechanisms</td>
</tr>
<tr>
<td>Neuron/Connection Slab mapping</td>
<td>block</td>
<td>2D topology for SIMD type Processor memory</td>
</tr>
<tr>
<td>Net/Net-module replication</td>
<td>process</td>
<td>2D topology for SIMD type Processor memory</td>
</tr>
</tbody>
</table>

Table 4.2 - Basic Mapping Schemes & Machines Features

It is firstly the level of parallelism the machine implements which determines the suitability of a mapping scheme. Secondly, the topology, communication and processors assess the feasibility of the basic mapping schemes.

Multiple mapping schemes are defined as combinations of basic mapping schemes. Following this, the suitability of the multiple mapping schemes can be determined by examining the suitability of the basic mapping schemes each consists of. The main issue is the comparison between the various multiple mapping schemes, i.e., which criteria to use to identify the best mapping scheme to implement. This issue is solved by considering the elements used to define the mapping schemes. The Abstract Decomposition module has devised the set of decomposition schemes on the basis of the type of neural systems' parallelism they exploit. These types, determined in Chapter 3, relate to the components forming a neural system: connection, neuron, slab of neurons or connections and system. The combinations, i.e., the set of multiple mapping schemes, can thus be evaluated by examining which elements of the neural system they map, and by evaluating the respective benefit. The next section, which describes the process of selecting a mapping scheme, considers this issue in more detail.

4.3.2.2. Selection Process

Based on the above analysis of mapping schemes relative to parallel machines, this section develops a selection process for identifying the most suitable mapping scheme. It defines, at a conceptual and informal level, the series of steps to follow to identify the most suitable mapping scheme according to the targeted neural system and parallel machine. The selection process identifies the most suitable mapping scheme on the basis of the types of parallelism exhibited by the neural system, it exploits. That is, among the alternative mapping schemes of a neural system, it selects the scheme which implements all the existing degrees of parallelism, if these parallelisms are exploitable on the targeted parallel machine. The process operates in two stages: firstly a basic mapping scheme is selected, this is referred to as the initial scheme, secondly the most suitable associated
multiple mapping scheme is selected. An associated multiple mapping scheme combines
the basic scheme with the exploitation of one or more types of neural system parallelism.
As stated in the previous section, multiple mapping schemes are determined by looking
at all the possible combinations of the initial scheme with the remaining schemes. For
example, from an initial neuron mapping scheme the set of multiple mapping schemes
available is obtained by combining the neuron scheme with all other available mapping
schemes, e.g. connection, neuron slab, connection slab. The selection of the most
suitable multiple decomposition scheme is based on both parallel machine features and
characteristics of the neural network system, and more specifically the size of the neural
system. This acts in two sub-stages. Firstly, the parallel machine is examined to extract
which multiple mapping schemes are suitable. Secondly, the neural system size and the
processing parallelised are examined in order to identify the most beneficial mapping
scheme. The scheme added to the initial scheme is simply the one offering the highest
number of parallelised components and operations. For example, in the case of an initial
neuron mapping scheme on an SIMD type parallel machine, the choice between the
combination with a connection scheme or a net replication scheme depends on the size of
the whole neural system versus the number of connections, as well as on the number of
times the system processing is performed versus the number of time the connections
processing is performed. The diagram below, Figure 4.12, summarises how the selection
process operates, in conjunction with the machine features and the information obtained
from the Abstract Decomposition module.

**Basic Schemes from the Abstract Decomposition**

![Diagram of mapping schemes selection process]

*Figure 4.12 - Neural System Mapping Schemes: Selection Process -
Among the basic decomposition schemes available, a basic mapping scheme is
firstly selected in function of the parallel machine. Following this, additional types
of neural systems parallelism are included, if their implementation matches the
characteristics of the parallel machine, and the neural system size.*
The selection process presented above is intentionally kept at a conceptual level. The MDD module is generic, it applies to a range of both neural systems and parallel machines, and thus had to be kept at such a conceptual level. Furthermore, as we shall see in the next chapters, this selection process is described in more detail, firstly via the implementation of the mapping software, and secondly via the development of an analytical tool to formalise the selection process.

4.4. Summary

This chapter has developed a System for decomposing and mapping neural systems onto general-purpose parallel machines. The System divides the task of mapping into a machine independent task; the Abstract Decomposition module (AD), and a machine dependent task; the Machine Dependent Decomposition module (MDD). In doing so, the System is able to cope with a range of neural systems and general-purpose parallel machines. It provides the mechanisms for the identification and specification of a near optimal mapping scheme for implementing a neural system onto a parallel machine.

The AD module analyses the parallelism exhibited by neural systems and identifies alternative decomposition schemes. It describes a specification for neural systems that makes explicit their various types of parallelism. The determination of decomposition schemes is based on the types of parallelism the schemes exploit. Decomposition schemes, both basic and multiple are specified in terms of their data decomposition and the processing phases parallelised by each scheme.

The MDD module evaluates the alternative mapping schemes of neural systems relative to the targeted parallel machine, and generates the parallel specification of the elected scheme. It developed an informal selection process for assessing the alternative mapping schemes of a neural system onto a parallel machine. Based on this, the MDD indicates the most suitable mapping scheme for the targeted neural system and parallel machine. The issues involved in the automatic generation of parallel specification of a neural system mapping are considered in the next chapter which describes the implementation of such a system.

To assess the System for mapping, a prototype mapping system for MIMD machines has been developed. This prototype analyses constructed neural systems and automatically generates their parallel specification for a representative example of MIMD machines; namely the SN1000 Transputer-based machine. It is generic in the sense that it applies to virtually any neural system, but is restricted to MIMD distributed memory machines.
Chapter 5

Prototype for MIMD Machines

This chapter describes a software prototype for mapping static neural systems onto Multiple Instruction Multiple Data (MIMD) machines. The design of the software is based on the System for decomposing and mapping developed earlier. The prototype was constructed within the Esprit project Pygmalion, and implements an automatic translation from the Pygmalion nC neural network specification to its equivalent on a Transputer-based machine; namely the Parsys SN1000.

5.1. Objectives

This chapter describes a mapping software prototype which automatically decomposes and maps static neural systems onto a typical MIMD machine, the SN1000 Transputer-based machine. The design of the prototype software is based on the System for decomposing and mapping neural systems onto parallel machines developed in Chapter 4.

The rationale behind the choice of Transputer-based machines as targeted machines is two-fold: firstly Transputer-based machines correspond to a realistic implementation of the Multiple Instruction Multiple Data (MIMD) concept associated with a pure distributed memory model, and secondly the large interest that such machines have received as target machines for the execution of neural systems [Bey89a], and [Ern90a], and [Joh90a]. Obviously, its accessibility at the university was also a criteria in the decision. Several researchers have implemented neural systems on Transputer systems. However, existing systems have not developed special automating facilities to help with mapping onto Transputer-based machines. They generally present the implementation of a particular neural system onto a particular Transputer-based machine [Zha90b]. There is as yet no system which is sufficiently generic to support the mapping of a range of neural systems. The prototype is able to adapt to various neural systems, in this respect it corresponds to a generic mapping software.

The construction of the mapping software prototype has two objectives:

- To verify the feasibility of an automatic mapping of neural systems onto parallel machines.
- To validate the proposed System for decomposing and mapping neural systems onto general-purpose parallel machines.
The prototype is able to automatically map a range of static neural systems, such as Back Propagation, Kohonen Self-organising Map onto a Transputer-based machine. The emphasis is put on the process of transforming a specification to its equivalent on a distributed memory parallel machine. This type of translation is recognised to be the hardest because requiring distinctions between computations and communications [Sab88a]. The design of the prototype corresponds to the proposed System for decomposing and mapping neural systems. It consists of an Abstract Decomposition module which decomposes neural systems irrespective of the targeted parallel machine, and a Machine Dependent Decomposition module which then generates the parallel specification of an optimal mapping scheme onto the targeted machine.

This chapter is composed of five sections. This section has declared the motivations and has defined the objectives of the implemented mapping software. The second section gives an overview of the Pygmalion project for which the prototype was developed. The third section presents the design of the mapping software and specifies its Abstract Decomposition and Machine Dependent Decomposition modules for decomposing and mapping neural systems onto MIMD machines. Following this, section four describes the implementation of the prototype and details the operations involved in its translation onto the Parsys SN1000 machine. Finally, the last section summarises the prototype and its outcome.

5.2. PYGMALION Project

The prototype presented in this chapter was developed as part of the Esprit PYGMALION project [Tre89c]. The prototype is a component of the PYGMALION Neural Network Programming Environment (NNPE), as shown in Figure 5.1. The Pygmalion NNPE comprises five major parts:

- **Graphic Monitor**, a graphical software environment for controlling the execution and monitoring a neural network application simulation.

- **Algorithm Library**, a parameterised library of common neural networks, written in the high level language and providing the user with a number of validated modules for constructing applications.

- **High Level Language (HLL) N**, an object-oriented programming language for defining, in conjunction with the algorithm library, a neural network algorithm and application.

- **Intermediate Level Specification (ILL) nC**, a neural network specification for representing the partially trained or trained neural network applications.

- **Compilers** to the target UNIX-based workstations and parallel Transputer-based machines.
The user of the Pygmalion environment specifies its neural application using the High Level Language (HLL) provided with an algorithms library. This specification is then translated into nC, the Intermediate Level Language (ILL). Finally, the Graphic Monitor enables the user to run his/her application interactively. The application can be either executed on a Unix-based workstation or on a Transputer-based machine.

One of the objectives of the Pygmalion NNPE was to interface to parallel machines such as Transputer-based machines. This would enable a programmer to generate neural network simulations that execute on both sequential machines and Transputer-based machines. Furthermore, the goal is to provide the same functionality for a neural network simulation executing on a host workstation (sequential) and on a Transputer-based machine (parallel). That is, the mapping software aimed not only to automatically generate parallel simulations but also to give its user the same monitoring facilities as for a sequential simulation.

Within this thesis framework, the emphasis is put on the automatic transformation of a neural system into its parallel equivalent. As such, it is not concerned by the facilities offered to users. It describes the mapping software for MIMD distributed memory machines, and details the implementation of the proposed System onto the Parsys SN1000.
5.3. Mapping System

As indicated by the System for decomposing and mapping developed in the previous chapter, the mapping software is composed of two modules: an Abstract Decomposition module (AD) and a Machine Dependent Decomposition module (MDD). The AD identifies decomposition schemes regardless of the machine\(^5\).

The MDD reduces the space of alternative mapping schemes according to machine characteristics, and provides the parallel specification of the optimal mapping scheme for the neural system onto the SN1000 Transputer-based machine.

Transputer-based machines are representative of MIMD distributed memory machines. The prototype is thus generic for MIMD distributed memory machines [Aze91a]. The Machine Dependent Decomposition evaluates mapping schemes, this is based on the framework presented in the Chapter 4. The MDD module constructed here implements a basic version of the framework. Indeed, the objective is to construct a system able to automatically map static neural systems. Thus, the emphasis is put on the translation task rather than on the analytical evaluation of speedups which is the subject of the next chapter. Furthermore, the Pygmalion project required rapid development of the prototype. Therefore, the MDD simply devises, off-line a set of rules to control the selection of the mapping scheme to be implemented.

A neural system running on a MIMD distributed memory machine can be described as a set of independent processes that communicate with each other via channels, i.e. Communicating Process (CP). It should be noted that CP is related to the notion of Communicating Sequential Process (CSP) developed by Hoare [Hoa78a]. They differ in that a CP may contain processing that can be parallelised, as opposed to a CSP in which the processing is purely sequential. Two issues are addressed when decomposing and mapping neural systems: a logical issue and a physical issue. The logical issue has to do with the specification of a neural system as a set of independent processes associated with the data exchanges they require. The physical issue discusses the actual mapping of this system onto the particular machine. The number of processors and their configuration are physical restrictions, while any number of communicating processes can be specified. These logical and physical phases correspond respectively to the operations performed by the Abstract Decomposition and by the Machine Dependent Decomposition modules. The following figure, Figure 5.2 displays the design of the mapping software for the SN1000 Transputer-based machine.

---

\(^5\) To the extent that it concerns MIMD distributed memory machines.
A neural system is firstly processed through the Abstract Decomposition module. The AD module operates a processing-based decomposition of the neural system. From this, alternative decomposition schemes are identified. These are evaluated by the Machine Dependent Decomposition which decides on the mapping scheme to implement. Finally, the parallel specification on the SN1000 Transputer-based machine is generated.

Decomposition schemes are specified as a collection of processes that can run in parallel (logical) phase. The Abstract Decomposition module extracts the processing and associated data requirements in order to specify the alternative neural system's decomposition schemes as CP-based systems. This is especially important because it is concerned with MIMD distributed memory machines for which communication and processing are distinguished. The analysis performed by the AD is processing-based, i.e. it identifies the processing phases that can occur in parallel and then extracts the associated data requirements. The MDD evaluates mapping schemes and provides the specification of the optimal one (physical phase). In the case of the prototype, the selection of a mapping scheme is performed on the basis of rules, devised off-line (during the design of the prototype). This section is divided into four parts. Firstly, the software used on the Parsys SN1000 is presented. Secondly, the Abstract Decomposition is described, whilst the Machine Dependent Decomposition is described in the third part. Finally, the integration of the AD and MDD is presented.

5.3.1. The Parsys SN1000

The Parsys SN1000 machine is used in conjunction with the 3L Parallel C & Tiny environment which controls both communication and synchronisation.

The basic information required to construct a program running on the SN1000 are: the code information, the configuration information, as shown in Figure 5.3. Source code is developed using the 3L Parallel C language and TINY interface. After compilation the binary is linked with the TINY library to produce a single processor executable code. Configuration information is specified by two components. The first component describes, within a high level configuration file, the channel connections and task
placement; specifying the location of the tasks onto the Transputer array. The second component specifies the Transputer network topology; describing the links between Transputers. These two components are then used by the high level configurer to produce: the switch file for the PARSYS machine, and the 3L configuration file. Finally, this configuration file and the single processor executable code are processed by the 3L configurer to produce the actual multi processor executable code.

**Figure 5.3 - TINY + 3L Parallel C Environment**

The Tiny + 3L environment comprises the 3L compiler, the high level configurer and the Tiny communication harness. The task source code combined with the Tiny interface are compiled to produce a single processor executable code. This code is then processed by the 3L configurer which, associated with the Tiny communication harness generates the multi-processor executable code.

**3L Parallel C** is based on an abstraction of Communication Sequential Process (CSP) [Hoa78a]. A 3L Parallel system is a collection of concurrently executing processes which can only communicate with each other via channels. Each of these processes are referred to as **tasks**, each task having its own region of memory for code and data. Furthermore, each task defines its input and output port vectors for communication purposes. A parallel program specified in 3L Parallel C corresponds to a set of tasks all potentially executable in parallel.

**Tiny** [Par90c] is a communication harness, developed by Edinburgh University. It has two components: a kernel, formed from configuration, routing table and message routing sub-components and an interface, formed from the Tiny tasks and the Tiny library. Communication between transputers is dealt with at a high level (specified in
Tiny tasks). Communication occurs between these tasks and not between channels, the
tasks do not specify which transputer channels are to be used. Tiny takes care of the
message routing and addressing problems. Tiny offers three types of message addressing
schemes: adaptive directed (non-blocking task to task communication), sequential
directed (blocking task to task communication), and undirected (non-blocking broadcast
communication).

When integrated, Tiny + 3L Parallel C offers a complete software package that
allows the development of parallel applications. Such applications are composed by a set
of concurrent processes; the 3L tasks, that communicate with each other through their
associated Tiny task, and the configuration specification that describes the actual location
of each of the previously defined tasks.

5.3.2. Processing-based Abstract Decomposition

The Abstract Decomposition module performs a processing-based decomposition. It
analyses the hierarchy of computations of a neural system to extract the types of
parallelism, and thus to identify the alternative decomposition schemes (cf Chapter 4:
Identification of decomposition schemes). The analysis of the computations considers
two classes of processing: processing that decompose further, referred to as higher, and
the processing that act on data, referred to as lower. Higher processing is found at the
system level (cf system processing), and also at the model level (cf global computations
of a model). Lower processing is found at the model level (cf Chapter 4). Each
processing indicates if their execution can be performed in parallel or in sequence.
Furthermore, each processing specifies the components it acts on. Alternative
decomposition schemes of neural systems are then specified as a set of communicating
processes, tasks in this case. This section is divided into two parts. Firstly, the method
for automatically decomposing neural systems is described. Secondly, the alternative
decomposition schemes are discussed.

5.3.2.1. Decomposing

The decomposition of neural systems is performed by identifying higher
computations and their execution type (sequential or parallel), and by identifying the
lower computations they are made of. Sequential higher computations and their
decomposition are firstly described. Secondly, parallel higher computations are
considered. Finally, a simple illustration of the processing-based decomposition and the
resulting set of communicating tasks is presented.

Sequential higher computations identify a set of subsidiary computations which
execute in strict sequence. An example of a process $S$ which decomposes into the
subsidiary processes $S_1$, $S_2$, $S_3$ is shown in Figure 5.4. The execution of $S_2$ can only
occur when the execution of the S1 has been completed. This type of computation forms a control task that acts as an execution scheduler.

Parallel higher computations identify a set of subsidiary computations that can execute in parallel. These computations can thus be distributed over a number of control tasks. For example, a parallel computation P composed of P1 and P2, as shown in Figure 5.5, indicates that the execution of P1 and P2 can occur in parallel. Therefore, the computation P and its two subsidiary P1, P2 are distributed over three control tasks to achieve parallel execution of P1 and P2. This concept of distributing the control is generalised and applies to any parallel higher computation.

Figure 5.5 - Example of a Parallel Computation P -

Figure 5.6 illustrates the decomposition of a simple set of higher and lower computations. The hierarchy is depicted as a tree in which the leaves correspond to lower computations, i.e. acting on data. Parallelism, in the figure, is expressed via parallel/sequential tree branches.
Figure 5.6 displays a computation tree and the result of its processing-based decomposition. As shown in the left-hand side of the figure, a higher computation R1 divides into R2 and R3. Both R2 and R3 divide into a set of lower computations that can execute in parallel. The AD identifies a processing-based decomposition in which R1, R2 and R3 form one task; CP123 in the figure, while all lower computations, i.e. from R21 to R34 correspond to independent tasks able to execute in parallel.

5.3.2.2. Neural Systems Decomposition Schemes

As described in Chapter 4, the alternative decomposition schemes of a neural systems are basic schemes, respectively connection, neuron, slab decompositions and net replication, and their combination. Each decomposition scheme exploits one or more types of parallelism. The processing-based decomposition generates each scheme by associating a task with the component of the system which exhibits the targeted type of parallelism. For example, in the case of a neuron decomposition (exploiting the intra-slab parallelism), a task is generated for each neuron of a same slab. Then, higher computations are decomposed accordingly. Each decomposition scheme is specified as a set of 3L tasks which communications are controlled via the Tiny tool. The table below, Table 5.1 lists the decomposition schemes, and their attributes.
Table 5.1 - Decomposition Schemes on MIMD machines -
The basic neuron scheme exploits the intra-slab parallelism by decomposing neurons of a same slab. The basic slab scheme exploits the inter-slab parallelism by decomposing slabs of neurons/connections. The basic network scheme exploits the training or net-module parallelism by replicating the entire network (or module). Multiple decomposition schemes correspond to the combination of the three basic schemes. This defines four multiple decomposition schemes respectively combining the neuron and slab scheme, the neuron and net schemes, the slab and net scheme, and finally the neuron, slab and net schemes.

<table>
<thead>
<tr>
<th>Decomposition Scheme</th>
<th>Exploited Type of Parallelism</th>
<th>Distributed Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>neuron scheme</td>
<td>intra-slab</td>
<td>neuron</td>
</tr>
<tr>
<td>slab scheme</td>
<td>inter-slab</td>
<td>slab</td>
</tr>
<tr>
<td>network scheme</td>
<td>training or net-module</td>
<td>system</td>
</tr>
<tr>
<td>neuron &amp; slab scheme</td>
<td>intra and inter-slab</td>
<td>neuron &amp; slab</td>
</tr>
<tr>
<td>neuron &amp; net scheme</td>
<td>intra-slab and training</td>
<td>neuron &amp; system</td>
</tr>
<tr>
<td>slab &amp; net scheme</td>
<td>inter-slab and training</td>
<td>slab &amp; system</td>
</tr>
<tr>
<td>neuron &amp; slab &amp; net scheme</td>
<td>inter, intra-slab and training</td>
<td>neuron, slab and system</td>
</tr>
</tbody>
</table>

Table 5.2 lists the alternative decomposition schemes of neural systems onto MIMD machines. It also indicates the type of parallelism they exploit, and the basic element of the neural system which is distributed. In the case of the prototype for MIMD machines, the connection decomposition is ruled out from the set of decomposition schemes. The reason for this is that the grain of a connection decomposition $g_c$ (as defined in Chapter 4) is far too large to expect any substantial speed-up from its implementation. And this, without taking into account the communication load. The set of decomposition schemes is thus reduced to neuron, slab decomposition, network replication and their combination. In the case of a network scheme, the entire neural system, i.e. model, application, and system processing, is associated with a single task. Each task replicates the model and acts on part of the application. The exchange of data occurs at the end of a learning phase so as to collect the adaptation of the weights of each system. The neuron decomposition scheme associates each neuron of a slab with a task. The connections weights are then associated on a neuron basis. In this case, data exchanges occur at each sequential step between each parallel neuron computation as specified by the neural system. The slab decomposition scheme associates each slab (from the set of slabs that perform in parallel) with a task. Data exchanges are similar to the previous case, i.e. they occur at each sequential step, but at a higher level, i.e. not between neurons/connections but between slabs of neurons/connections.

The multiple decomposition schemes available are based on the combination of these basic schemes. The combination of neuron and slab decompositions (case where a slab represents a set of neurons) specifies that no overlap are allowed between the tasks representing neurons of different slabs. That is, each neuron of different slabs are associated with a task and their groupings is controlled by higher level task in order to
achieve parallelism along slabs. Data exchanges comprise the exchanges for neuron and slab decomposition schemes.

5.3.3. Machine Dependent Decomposition

As indicated at the beginning of this section, the emphasis is put on the automatic generation of parallel specification onto an MIMD distributed memory machines. For this prototype, the rules for selecting an optimal mapping scheme for a neural system are devised off-line. In other words, a set of rules has been devised during the design of the mapping system, and the implementation follows them, in a static manner. The characteristics of MIMD machines are analysed in conjunction with the characteristics of the neural decomposition schemes. The following set of informal <if conditions - then mapping selection> is devised:

```markdown
if (machine size << model size )
    if (application size >> model size) & (processor can contain a whole net)
        then select the net replication scheme
        if (processor can not contain the whole net)
            then select combination of neuron & slab mapping scheme
    if (application size ∼= model size)
        then select combination of slab mapping & net replication scheme
    if (application size << model size)
        then select combination of slab and neuron mapping scheme
if (machine size ∼= model size)
    then select combination of slab mapping and net replication scheme
```

Firstly, the machine size \( p \) is evaluated against the neural model. The neural model is represented by \( N_s, N_i, N_c, N_p \), \( N_s \) the number of slabs, \( N_i \) the number of neurons, \( N_c \) the number of connections, \( p \) represents the number of processors of the machine. If the model size is equivalent to the machine size, then the combination of slab decomposition and net replication is selected; \( \Theta_p \) versus \( \Theta_{p,r} \). Otherwise, the neural application size is considered. The application size is represented by \( N_{ip} \) number of training patterns. If the application size is large relatively to the model size, then according to the processor size, the network replication scheme is implemented. In this case, the processor size \( PE_i \) must be able to contain the whole neural system. Otherwise the combination of neuron and slab mapping is selected. Communication mechanisms are also analysed, external communications are considered with the neural application size \( N_{ip} \), especially in the case of network replication. Indeed, inter-processor communication is simply used to collect the weight adaptation in which case broadcast mechanisms are used if available on the machine. If the application size is equivalent or smaller than the model size, then
neuron, slab decompositions and network replication are compared; \( \Theta_n, \Theta_s, \Theta_p \). Internal communications mechanisms \( (\Theta_i) \) are analysed with the neural model to evaluate the overload created by a neuron or slab mapping. Furthermore, in the case of a combination of slab decomposition and net replication, the mapping creates clusters of Transputers which are then associated with each particular replicated neural system. The external communication is used to load each copy of the model and its part of the application. Furthermore, when investigating the parallelism exhibited by neural systems, we concluded that training parallelism, even though present in certain models, has not been fully analysed. This implies that the consequences of exploiting this parallelism are in a way unknown. Thus, it is necessary to consider situations where this type of parallelism is not present. In these cases, it is the neuron and/or slab mapping schemes which are implemented.

5.3.4. Mapping Software & the SN1000

The mapping software integrates the Abstract Decomposition and Machine Dependent modules to produce the SN1000 parallel specification of neural systems. As shown in Figure 5.7, the prototype acts on the nC memory image of a neural system.

![Figure 5.7 - From A Neural Network System to its Transputer-Based Machine Specification](image)

Having constructed a neural system in nC, the mapping software provides its processing-based decomposition along with the necessary data exchange. Based on this, it then generates the appropriate SN1000 specification: the task source code, the task configuration, and the transputer topology.

To generate the Transputer-based machine parallel specification of a neural system, the mapping software implements a processing-based decomposition of the nC structure, and automatically generates the tasks' code and configuration. As described in the next section, the extraction of the data exchanges between tasks is enabled by the extensions provided to the nC specification. The recommendations made by the Machine Dependent Module are integrated within the processing-based decomposition, which

---

\(^6\) The main reason for this is the lack of consideration of neural systems' parallelism.
creates the appropriate tasks for the selected mapping scheme. The multi processor executable code associated with the transputer switch file is finally generated from the task source code, the task configuration, and the Transputers network connections. Furthermore, it should be noted that the implementation of the mapping software incorporates the AD module with the MDD module. That is, the decomposition of the nC rules structure takes into account the recommendations given by the MDD, which then generates the appropriate code. In summary, the translation from nC to the SN1000 Transputer machine generates three elements:

- Task code, (<file>.c) which is made of 3L Parallel C code and driven by message passing.
- Task configuration, (<file>.hlc) which specifies the task types, the task grouping and placement onto the processors.
- Transputer configuration, (<file>.map) which defines the topology of the Transputers network.

5.4. Implementation

This section describes the implementation of the mapping software carried out in the Pygmalion project. It comprises three parts. The first part presents the extensions to the nC specification that have been proven necessary. The second part describes the automatic generation of the SN1000 specification by the mapping software. Finally, the third part presents some experiments.

5.4.1. nC Extensions

The explicit control of execution given in nC does not provide enough information to guide the mapping software for translating nC neural systems onto MIMD distributed memory machines. For example, the rule_init and rule_extent for the neuron's state_update rule, shown in Figure 5.9 and 5.10, specify the data the rule needs but does not indicate which data are local to the rule and which data are global. Moreover, this cannot be retrieved from the body of the rule because the body is generic and as such applies to an unknown list of parameters. The data exchanges required between rules are thus non identifiable. To overcome this, two basic mechanisms are added. They concern the rule building procedure and add information to the rule structure. In doing so, they provide the prototype with the necessary information for automatically producing the SN1000 specification.

The extensions introduced are in the form of a keep-init and keep-extend mechanisms. They are analogous to the rule-init and rule-extend mechanisms in that they instantiate the information each rule embeds. The information determines for each existing rule its
data requirements. More specifically, they detail the data required for execution by each rule, or the data required for the execution of further rules. In the case of a rule acting on data; e.g. a rule addressing a neuron, the keep_init and keep_extend lists define the data the rule needs to send back. These data correspond to data local to the rule. In the case of a rule acting on sub-rules; e.g. a rule addressing a cluster, the keep_init and keep_extend lists define the data the rule needs to send down to the sub-rules. These data correspond thus to data non local to the concerned sub-rules. Having these information, the analysis of the rule structure is able to extract the data each rule needs to send up or down. These data requirements, held in the rule, are specified in the form of symbolic tags which describe the concerned element within the nC data structure of a neural system. The figure below, Figure 5.8, presents the keep_init list for a neuron error calculation rule.

```
  pointer to rule
    initialisation | extension
    -------------------
      n_0_*_0_*dl (symbolic tag)

/* snode addition */
/* send back the error of the concerned neuron */
sprintf(buffer, "n_0_*_0_*dl");
keep_init(&sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->synapses,
    0,
    buffer,"null",
    EOP);
```

**Figure 5.8 - An Example of Keep-init Mechanism**

The rule representing the neuron error calculation requires to send back its error. The symbolic tag \texttt{n\_0\_*\_0\_*dl} indicates that for any layer (first star character) and any neuron (second star character), the first local data (dl) to the neuron is to be sent back.

The keep-init and keep-extend mechanisms allow the mapper to extract the data requirements for each rule. The syntax to represent these requirements follows the hierarchical nC structure of a neural system. Data items are symbolically represented by a tag which describes their path within the nC structure. For example, \texttt{n\_0\_1\_0\_2dl} refers to the second data item of the third neuron of the first hidden layer in a neural system made of one net with no concept of cluster. A primitive set of syntactic rules has been devised to represent all variations, they are not detailed here. The syntactic rules are such that: the character "*" indicates that any instance of the level addressed is to be send back from the current rule, whilst it is the character "A" which indicates that any
instance of the level addressed is to be send down to the sub-rules. Figure 5.9 shows the keep_init mechanism for a rule that decomposes into further rules.

```c
/* snode extension */
sprintf(buf, "n_0_%d_0_Ad0", (i-1));
keep_init(
    &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_state_upd],
    0, buf,
    EOP);
```

*Figure 5.9 - Keep-init Mechanism*

The rule representing the cluster state update requires to send down the state of all the neurons within the previous cluster (i-1) as described by the symbolic tag n_0_%d_0_Ad0.

These symbolic tags are used when the mapping software is generating the actual code and the data exchanges. An example of a neural system specified in nC which incorporates these extensions is given in Appendix 5.1 which gives the build_rule procedure of backpropagation neural systems.

5.4.2. Automatic Generation

The constructed prototype is integrated within the Pygmalion Programming environment; as such the control of the high transforms; i.e. learn and recall of neural systems is given to the user of the environment. The high level transforms are thus not specified within the nC rules but are specified by the user (through the main program). Following this, it is was very difficult to retrieve the information necessary for the specification of a network replication scheme. Indeed, the system-related type of parallelism, which indicates that patterns can be given to a neural system independently, can not be extracted. Therefore, the implementation is restricted to neuron and slab decompositions schemes, and their combination. This section firstly describes the analysis of the nC rule structure. Secondly, it specifies how the data exchanges are automatically generated. Finally, it describes the automatic generation of the tasks' code, configuration, and Transputer topology.

5.4.2.1. nC Rule-based Decomposition

As described in section 5.3.4, the mapping software performs a processing-based analysis. In the case of the nC specification, it analyses the rule hierarchy and extracts the types of parallelism to decompose the neural system. Two classes of rules are considered: the rules that decompose into further rules called meta rules, and the rules that act on data called ground rules. Each rule indicates its execution type. These execution types typically specify a parallel (pexec) or a sequential (sexec) execution. Each rule specifies the components it acts on, as part of its parameters list, using both the para_list and keep_init lists. The meta rules typed as sexec, identify a set of subsidiary
rules which execute in strict sequence. This type of rule (sexec) forms a control task, referred to as Controller, that acts as an execution scheduler. The meta rules typed as pexec indicate where parallelism can be performed and decomposes into parallel execution of further rules. These rules are thus distributed over a number of Controller tasks. Ground rules simply generates their equivalent in 3L Parallel C & Tiny, according to the distribution they are or not included within Controller tasks.

To accommodate more complex rule execution pattern (e.g. looping), variations of the basic sexec and pexec rule types have been introduced. A meta rule typed as [p,s]exec_r indicates an iterative execution (i.e. r stands for repeat) of its rule components. That is, the list of rules forming the meta rule is executed cyclically until one of the components (a rule) returns a termination code. For example, in the backpropagation system a pattern has to be learnt until the tolerance test is successful. Furthermore, to allow any arbitrary nesting of meta rules, a type [p,s]exec_c has been added (where c stands for clear). This execution type only differs from the previous in that it does not propagate the termination code up through all the nested rule levels. Taken together, those two types accommodate any kind of rule execution flow. Fortunately, these added execution types do not alter the methodology. As described above, sexec, sexec_c, sexec_r rules form one control process, while pexec, pexec_c, pexec_r rules are distributed over a number of control processes.

The following figure describes this rule-based decomposition applied to the learning operation of backpropagation neural system for which the nC code is given in Appendix 5.1. Controller tasks are generated from the analysis of the nC memory image and associated rule hierarchy.
Four Controller tasks are generated: a Controller which controls the learning computation, a Controller which controls the recalling computation, a Controller in charge of the Layer-Weight_Update for the hidden layer, and a Controller in charge of the Layer-Weight_Update for the output layer. It is the learning phase controller which controls the parallel execution of the controller 3 and 4, and their correct termination before the execution can proceed any further. The automatic generation of data exchanges between Communicating Process is now detailed.

5.4.2.2. nC Data Exchanges

In nC, ground rules parameter list indicates which data are required by the rule and which data the rules needs to send back, via its keep-list and keep-extend mechanisms. It also indicates which components of the nC data structure hold the data. For example, a ground rule specifying a neuron state update consists of a list of data to be supplied to the rule function. This data comprises: - the neuron's state parameters, - a set of synapse parameters, defining all the incoming weights for the neuron, - a set of neuron parameters, defining the states of the input neurons, and - a net parameter defining the learning rate. The analysis of these rule parameters identifies the data which each ground rule needs (in order to operate), and also the location of each data within the nC data structure. This defines the inter-dependencies between ground rules.
For a ground rule in the nC rules hierarchy, the master data correspond to local data on which the ground rule operates. The slave data correspond either to data which have to be obtained from another ground rule, or to parameters located at a certain level within the nC data structure. It should be noted that slave data for a rule G1 can be a master data for a rule G2. For a meta-rule, the master data correspond to local data set up at initialisation (e.g. learning rate), the slave data correspond to the master data of its sub ground rules.

The following specifies the generic data communication exchanges. Exchanges between meta-rules are firstly considered. Then, data exchanges between meta and ground rules (and vice versa) are presented. In the following, the buffer of data is filled according to the information held in the keep_init and keep_extend lists of the rule (as described in section 5.5.1.2).

**Data exchange between meta rules**

They are specified by the sigla MM

- a meta X sends an order (start, end) to another meta Y
  
  `MMsend_exec(to, order)`
  
  `MMsend_exec(int, int)`

- a meta X waits for an order (start/end) from another meta Y. This occurs only if the meta X has no data to send to the meta Y, otherwise the receiving of the data acts as the order. This checks if the source of the message received is legitimate by screening the from_array (which holds all legitimate association taskID<->meta_tag (see Appendix 5.2))
  
  `MMwait_end(from_array, length)`
  
  `MMwait_end(int[], int)`

- a meta X sends data to another meta Y. This occurs only if the meta X has data to send to the meta Y; i.e. they are not part of the same Communicating Process.
  
  `MMsend_data(destination, buffer, length_of_buffer)`
  
  `MMsend_data(int, float[], int)`

- a meta X waits data from another meta Y
  
  `MMwait_data(from, buffer, length_of_buffer)`
  
  `MMwait_data(int, float[], int)`

**Data exchange between meta and ground rules**

This is specified by the sigla MG.

- a meta X sends an order to ground G. This is similar to MMsend_exec except that ”to” has to be found from the desti_gen_x array which hold all the legitimate association taskID<->ground_tag)
  
  `MGsend_exec(to, order)`
  
  `MGsend_exec(int, int)`

- a meta X sends data to ground G. This occurs only if the meta X has data to send to the ground G. Similar to MMsend_data except that ”to” has to be found from the desti_gen_x array, array which holds the taskID<->ground_tag associations.
MGsend_data ( destination, buffer, length_of_buffer)
MGsend_data ( int, float [], int )

a meta X waits data from grounds G. This occurs only if the meta X has data to send to G1... Gn, and if meta X is typed as "pexec"

MGwait_data( thearray, buffer, length_of_buffer )
MGwait_data( int [], float [], int )

Data exchange between ground and meta rules
This is specified by the sigla GM

a ground G sends an order (exec/end) to a meta X. This occurs only if the meta X has no data to send to the ground G; i.e. the ground G does not wait for data from X.

GMsend_correct ( to )
GMsend_correct ( int )

a ground G sends data to a meta X. This occurs only if the meta X has to wait for data from the ground G.

GMsend_data (to, buffer, length_of_buffer)
GMsend_data (int, float [], int )

a ground G waits data from a meta X. This checks if "from" is equal to the source of the message received

GMwait_data (from, buffer, length_of_buffer)
GMwait_data (int, float [], int )

5.4.2.3. Tasks Code & Tasks Configuration

As indicated by the rule-based decomposition, the mapper produces two types of tasks; a Worker type, that processes data, and a Controller type that controls the simulation. Controller tasks are conceptually analogous to the meta-rules, while the Worker tasks are analogous to the ground-rules. More specifically, the Controller tasks are composed of meta-rules and implement the parallelism. That is, they dictate which group of Worker tasks is to execute in parallel and control their termination. Furthermore, we have explained how the Tiny communication harness takes care of the routing of the messages. Each task has a unique identifier, the taskID which is used for inter-tasks communications. These taskID are created at run-time, they are assigned according to the order of creation of the tasks; the first task created has a taskID equal to 0, the second task created has a taskID of 1, and so on. This has allowed us to devise a simple algorithm which creates and maintains a task-ID versus task lookup table for intra-tasks communication. Its code is given in Appendix 5.2, and takes in the neural model size (number of slabs, neurons per slab, connections) and neural application size (number of training patterns) and generates a table look-up of existing tasks and their taskID. Each task refers to this table when it wants to communicate with other tasks, allowing each task to know which task-ID it should send its message to.
Worker tasks

There is a small set of generic types for instantiating Worker tasks. Each generic type is associated with a particular neuron type, while each Worker task is associated with a particular neuron. A neuron type refers to the neurons that belong to the same net, layer, and cluster [Aze90b]. Therefore, a generic task has many instances, as defined by the network configuration. During the analysis of the nC rules hierarchy, the mapper creates a new generic task for each new type of neuron. The operations that the created generic task might then performed are the ground-rules associated with this neuron type. In conclusion, a generic task implements the functionality of a particular neuron type and is formed by one or a set of ground-rules. The code of such a generic task is as follows:

```c
/* Generic task for neurons route net 0, layer 1, cluster 0 */
for (;;) {
    switch (GMwait_exec(any_controller)) {
    /* Generic ground rule route 0 1 0 index I where there are some data to send back */
        case I:
            wait_data(data, meta_tag)
            process(Ground-rule Name)
            send_data(data, meta_tag)
            break;
        /* end generic ground index I */
    /* Generic ground rule route 0 1 0 index J where there are no data to send back */
        case J:
            wait_data(data, meta_tag)
            process(Ground-rule Name)
            send_end(1, meta_tag)
            break;
        /* end generic ground index J */
    }
}
```

The code of a generic task corresponds to an infinite loop in which the task is waiting for an execution order from a controller, e.g. GMwait_exec. According to the order received, the task executes the appropriate ground rule. If the execution of the ground rule does not require any data, it directly executes the ground rule, otherwise it firstly waits for the appropriate data.

Controller tasks

Controller tasks control the neural network simulation. A Controller task is created for each meta-rule or set of meta-rules that can be executed in parallel. If a set of meta-rules have to be sequentially executed then they are all incorporated within a unique controller task that controls their execution and their synchronisation in time. The overall control is provided by a special Controller task which sits on the interface Transputer.
The creation of Controller tasks is described by presenting the code generated by a sequential meta-rule (sexec type), and a parallel meta-rule (pexec type). An sexec-type meta-rule identifies a set of subsidiary rules which execute in strict sequence. Thus, this meta-rule and its subsidiary meta-rules are incorporated within a Controller task that acts as an execution scheduler. The components of a pexec type meta-rule can be executed in parallel. This type of rule decomposes into a parallel execution of further rules, which are then distributed over a number of Controller tasks. Such meta-rule and subsidiary rules \( P_i \) form \( x \) Controller tasks (\( x=1+ \) number of subsidiary rules), achieving the parallel execution of the \( P_i \) rules. Firstly, the code of Controller tasks that incorporate a set of sequential meta-rules are described. Secondly, the code of Controller tasks generated from a set of parallel meta-rules are presented.

(a) sexec meta

This describes, in an abstract manner, the code of a controller, referred to as Meta, of the sequential type which decomposes into \( m_1, \ldots, m_2 \) meta rules. The controller simply calls, in sequence, the procedures performing \( m_1 \) and \( m_2 \) and check their proper termination, i.e. these procedures send back their termination code (typically 0 for bad termination, and 1 for correct termination).

```c
int Meta() {
    if ( !m1() ) return(0) ;
    if ( !m2() ) return(0) ;
};
int m1() {
    execute /* return(0) if incorrect termination */
    return(1)
} ;
int m2() {
    execute /* return(0) if incorrect termination */
    return(1) ;
} ;
```

(b) pexec meta

This describes, in an abstract manner the code of a controller task, referred to as Meta, of a parallel type which invokes other controller tasks, referred to as \( M_1 \) and \( M_2 \). The controller sends execution orders to both \( M_1 \) and \( M_2 \), and data if required. It then simply waits to receive termination signal from \( M_1 \) and \( M_2 \), in any order (this depends upon which controller from \( M_1 \) or \( M_2 \) terminates first).
int Meta() {
    send_exec(M1, 1);
    send_exec(M2, 1);
    MMsend_data(data_buffer, M1 & M2);
    MMwait_end(M1 M2);
    MMwait_data(data_buffer, M1 & M2);
}

where M1 and M2 are themselves controller tasks with a code such as:
int M1() {
    MMwait_exec(Meta)
    MMwait_data(data, Meta);
    execute
    MMsend_end(Meta);
    MMsend_data(data, Meta);
}

The message passing mode used is asynchronous; i.e. non-blocking. Thus, each task starts processing as soon as a message arrives, and the sender does not wait for its message to be delivered but carry on its processing.

Tasks Configuration

The SN1000, under the 3L Parallel C + Tiny environment allows flexible processors network topology; i.e. the Transputer network topology is controlled by the user via the switch file. During this prototyping phase, it has been decided to use a fixed topology. The major reason of this is the cost in time when reconfiguring the network [Hol92a]. An hypercube like topology was chosen as the fixed topology. This topology implements a full virtual connectivity between Transputer devices. Thus, any task is able to communicate with any other task regardless of its type. The specification of the topology can be found in Appendix 5.3, this file <hypercube.map> is created once and for all. The configuration of the tasks onto the Transputer machine adapts to the variable number of tasks; either Worker or Controller tasks. Because the objective of this prototype is not to tackle the problem of mapping \( N \) processes onto \( P \) processors (known to be NP-complete [Bok81a]), the following method is used for their placement;

in case of a neuron mapping scheme the instances of a type of task (Generic or Controller) are positioned sequentially on the 48 Transputers. are placed onto the Transputer 1 to 48, and the remaining 4 instances are placed onto the Transputer 1 to 4.

in case of combination between neuron and slab mapping, the Controller tasks in charge of the slabs are located onto the Transputer which communicates with each neuron-CP tasks with a constant number of hops.
This method is incorporated within the mapping software, and allows the tasks to be configured at run-time. The task configuration file <file.hlc> is created automatically by the software, the code is given in Appendix 5.4. Each time a Controller or Worker task code is generated the software produces the associated task declaration. For example, in the case of creation of a generic Worker task, the following code is generated:

```c
Taskdef Gen010 {
    inputs=4 outputs=4
    data=500k taskfile="generic_gr_0_1_0.b4"
}
```

This code declares a task Gen010 made of 4 input and 4 output parameters, with a memory capacity of 500kbytes and which compiled source code is contained in generic_gr_0_1_0.b4 (the extension "b4" indicates a multi-processor executable code). Then, once all task codes have been generated (and thus their number is known) the software produces the associated task configuration declaration. Taking the previous example, it gives:

```c
/* declaration of the group of tasks containing the task Gen010 */
Group autom_0 {
    Task Gen010 : gen010_0 ;
    Task TripleTiny: tiny ;
    gen010_0 {
        in(*, *, tiny[3], taskId),
        out(*, *, tiny[3], type0) }
}

/* Position the group of tasks onto the network of Transputers */
Place {
    Root on [0]
    autom_0 on [1->10]
}
```

As shown above, the previously created Generic Worker Task Gen010 is now incorporated within a group of tasks and then positioned onto the network of Transputers.

In summary, the mapping software analyses the nC rule and data structure of a neural system and automatically generates the following:

- **<file.map>** - This file specifies the Transputer network topology. It is fixed and corresponds to an hypercube topology.
- **<file.hlc>** - This file declares the tasks, groups them, and positions them onto the Transputer network. It is created at run-time and in conjunction with the generation of the Controller and Worker task codes.
- **<file.c>** - These files contain the code of the Controller and Worker tasks. They are automatically generated from the analysis of the nC rule and data structure of the considered neural system.
5.4.3. Experiments

Within the Pygmalion project, the translation of the Competitive Learning and Backpropagation neural systems have been tested. These neural systems described in nC are automatically transformed into specifications suitable for execution onto the Parsys SN1000 machine under the 3L Parallel C + Tiny environment [Aze90c]. A short analysis of the performance of a backpropagation system implementing a standard application was carried out. The neural system uses a fully connected backpropagation model with a (96-24-96) topology. The neural application is a standard digit recognition application, in which the objective is to trained the neural system to recognise 3 digits, each digit is coded as an array of 8x12 pixels. The user of the Pygmalion neural environment interacts with the neural system via a global Controller task. This task is model-independent and allows a user firstly to specify which processing she/he wishes the neural system to execute, i.e. learn or recall, and secondly to enter the appropriate patterns via pattern handling software [Aze90b]. The user is also able to recover data from the neural system via this task. The format in which data are retained by the SN1000 and by the Pygmalion environment are equivalent, this allows the user to use the Graphic Monitor developed by the Pygmalion project. The automatic generation of the parallel specification of the neural system is described in Appendix 5.5.

The execution time for the digit application is about 1.5 faster on the Parsys SN1000 than on the Sun4 machine, this discounts the time spent in loading the patterns onto the parallel machine. On the Sun4, it costs 15s for the application to terminate, whilst it costs 10.6s on the SN1000. Other similar experiments have been carried out but the speed ups obtained were similar; they are thus not reported here. Justifications of these poor performances are two fold: firstly, the application used for evaluating the performances, and secondly the insufficient study of the 3L Parallel C + Tiny software before development of the mapping software.

Firstly, to achieve learning convergence for the digit recognition application, the neural system requires about 100 iterations over the set of patterns. This is quite small compared to real life applications for which the number of iterations over the set of training patterns is typically of the order of 10000. By considering these applications, the time spent in computation versus communication on the Parsys SN1000 is expected to increase quite substantially, and thus better speed ups are obtainable. In [Ref93a], the backpropagation model is used to assign rating to different stocks within a universe. In this example, convergence is reached within 30000 iterations, and typically requires 3 days of CPU time on a Sun workstation. It is estimated that running this application on the Parsys SN1000, a speed up of 10 is obtainable.

Secondly, the 3L Parallel C + Tiny package was studied after having constructed the mapping software. The inter processor communication time under Tiny was tested as
a function of the number of tasks per processor, and was also compared with other tools available (e.g. CStools), cf [Hol92a]. This investigation revealed a severe weakness in the Tiny tool: the number of tasks allocated per processor increases the inter-processor communication time quite drastically. The communication time for inter-task communication is 421 bytes per millisecond when there is one task per processor, and doubles if there are two tasks per processor [Hol92a]. This aspect was overlooked during the development of the prototype. The required number of tasks was created depending on the configuration of the model. Then, these tasks were grouped and positioned on specific processors, as specified by the Machine Dependent Decomposition (MDD). In doing so, a processor ended up hosting more than two tasks. In our example, processors can host three tasks: one controller and two workers. Hence, the inter-processor communication was not optimised. To overcome this, the MDD module requires to be modified so as to group different types of generic worker tasks, specified by the processing-based decomposition, within a same task. This would reduce the number of tasks each processor embeds. Furthermore, in doing so, the number of process switching, i.e. the number of times a processor has to change the task it is executing will decrease significantly, and also the execution time.

Lastly, it should be noted that the mapping software demonstrates the feasibility of an automatic generation of parallel neural system specifications. The software is also integrated with the Pygmalion environment, this allows a user to interact with the execution of a neural system in a similar manner whatever machine the execution occurs on (sequential or parallel). However, the construction of the mapping software did not optimise the usage of the 3L Parallel C + Tiny package. Another factor to consider is the slow speed at which the host communicates with the Parsys SN1000. This is due to the fact that the Parsys SN1000 can only communicate with the host via its root processor. Furthermore, the root processor, in absence of any operating system must interrupt the Unix OS in order to send a request to open and read a file on the host. This could be avoided by integrating the Tiny + 3L Parallel C software package with the Idris Operating System (OS). In doing so, the set of training patterns could be loaded onto the machine before any function is performed. Furthermore, this will permit the access to the internal disk of the SN1000. The internal disk could then used used to store the training patterns. In doing so, the implementation will avoid costly pattern reading operations.

5.5. Summary

This chapter described the implementation of a mapping software prototype which automatically decomposes and maps static neural systems onto the Parsys SN1000 Transputer-based machine. The design of the prototype is based on the System for
decomposing and mapping neural systems onto parallel machines developed in Chapter 4.

The system consists of an Abstract Decomposition module which decomposes neural systems irrespective of the targeted parallel machine, and a Machine Dependent Decomposition module which then generates the parallel specification of an optimal mapping scheme onto the Parsys SN1000. The prototype implementation analyses neural systems specified in nC and generates both the task code and task configuration necessary for execution on the Parsys SN1000. The AD module performs a processing-based decomposition of neural systems, i.e., the nC rule structure is analysed and rules that can execute in parallel are identified, their data requirements is automatically extracted. From this, the MDD module generates two types of tasks: Worker tasks that process data, and Controller tasks that control the execution and termination of groups of Worker tasks. The MDD also generates the task configuration. This includes both the task declaration and the task placement for the SN1000.
Chapter 6

Machine Dependent Decomposition: Analytical Framework

This chapter develops an analytical framework for the Machine Dependent Decomposition module. This additional tool is able to forecast the speedup of alternative mapping schemes onto various parallel machines, and thus select the scheme with the fastest speedup. This chapter also illustrates the usage of the analytical framework for SIMD machines, and specifically the Distributed Array of Processors (DAP). It demonstrates its validity by showing that the optimal mapping scheme of backpropagation systems analytically retrieved is the same as the one obtained experimentally.

6.1. Overview

The neural Machine Dependent Decomposition module described in Chapter 4 had two roles: firstly to evaluate the benefit of alternative mapping schemes of a neural system, and secondly to provide the specification of the optimal mapping scheme of a neural system. This chapter develops an additional tool which quantifies the evaluation of alternative mapping schemes of neural systems. It describes a framework able to analytically forecast the speedup of various mapping schemes of a neural system onto various parallel machines. The following figure, Figure 6.1 shows how this additional component integrates within the MDD module.
This chapter describes the analytical process for the selection of an optimal mapping scheme. In addition to this, it also demonstrates the usage and validity of this analytical process on SIMD machines, and specifically the Distributed Array of Processors (DAP). This is achieved by showing that the optimal mapping scheme analytically retrieved is the same scheme obtained experimentally. Firstly, the alternative mapping schemes of backpropagation neural systems onto the DAP are analysed according to the analytical framework. Secondly, the speedup of each mapping scheme is analytically expressed, allowing the set of schemes to be ranked. Then, these schemes are implemented onto the DAP, and their execution is timed. The analytical results are finally compared with the experimental results; proving the validity of the framework.

The rationale behind the choice of the Distributed Array of Processors (DAP) as the targeted machine is three-fold: i) it enables to cover the spectrum of general-purpose parallel machine, ii) it is a representative example of SIMD machines, iii) it is accessible. The DAP machine has been used in a variety of applications; ranging from image and signal processing, to data base searching [Col90a], and is known in the parallel community. Lastly, we had access to such a machine at the Computer Centre Queen Mary & Westfield College, from University College London through the JANET network.

The rationale behind the choice of backpropagation systems as targeted neural systems is that they are the most widely used. The backpropagation model has been applied to a wide range of applications, covering speech processing applications
Moreover, backpropagation neural systems have been extensively implemented on a variety of parallel machines [Par90b], and more particularly on SIMD machines [Wit89a]. This enables us to compare the prototype to existing parallel implementations. It is for these reasons that this chapter focuses on backpropagation neural systems.

This chapter is composed of four sections. The first section develops the analytical framework for evaluating the speedup of neural system mapping schemes onto virtually any parallel machine. The second section then describes the decomposition schemes of backpropagation neural systems. In the third section the analytical framework for the evaluation of speedups of backpropagation neural systems onto the DAP is specified, along with the software environment of the DAP machine. Section four describes the experiments conducted in order to validate the analytical results obtained. Finally, section five summarises the whole chapter, the analytical framework, its usage, and finally the experiments and their outcome.

6.2. Analytical Framework

6.2.1. Objectives & Basis

The objective of the framework is, from a set of alternative mapping schemes to decide which one is the optimal to implement. The optimal mapping is the one that offers the most rapid completion of the neural system; i.e. the mapping scheme $k$ with the best speedup $\Theta_k$, as expressed in (4.1) and (6.1). Efficiency $\epsilon$, as described in (6.3) is not directly addressed. That is, it would be possible to recommend a mapping scheme that utilises poorly the processors of the parallel machine as long as the mapping scheme is the one offering the best speedup. However, these two issues are interrelated, i.e. it is generally the case that a mapping scheme presenting the best speedup is the one that utilises the processors at best [Sev89a].

$$\Theta_{sk} = \max ( \Theta_{sk}, \Theta_{sk}, \Theta_{sk} )$$ (6.1)

$$\Theta_i = T_s / T_i$$ (6.2)

$$\epsilon_i = \Theta_i / \rho$$ (6.3)

where

- $\Theta_{sk}$ - the speedup factor for the mapping scheme $M_s$ onto the parallel machine $PM_k$
- $T_s$ - time for the neural system to execute sequentially,
- $T_i$ - time for the neural system to execute in parallel while implementing the mapping scheme $i$. 

101
number of processors. The speedup of a mapping scheme $i$ as defined in (6.2) corresponds to the factor by which the time for executing a neural system (implementing the mapping scheme $i$) is reduced. The ideal speedup (and maximum) for a parallel machine with $p$ processors is $p$. The efficiency (the ratio between the speedup and the number of processors) estimates the utilisation of the machine’s processors. The ideal efficiency is 1 and corresponds to a maximal speedup. In our case, the evaluation of the speedups $\Theta_i$ for a set of mapping schemes $i$ is static in that characteristics of both neural systems and parallel machines are predetermined. The neural system is constructed; the model used and its application are defined, and similarly for the considered parallel machine. The framework relates analytically mapping schemes and their associated exploitation of types of parallelism to the targeted parallel machine.

A low-level methodology for analysing the performance of multiprocessors architectures is the load/store kernels model [Gal89a]. This methodology is a systematic one in which the performance characteristics of a memory system is probed via a hierarchy of data-movement kernels. This technique has been used for the performance evaluation of parallel algorithms on specific parallel machines [Bod90a], and also for the estimation of various data distribution schemes on distributed memory multiprocessor [Bal91a]. The analytical framework is analogous to this technique. It assumes that performance predictions can be expressed in terms of the behaviour of basic templates. These basic templates determines a machine relative to the mapping schemes of neural systems. The load/store kernels methodology is "extrapolated"; i.e. instead of using low-level templates to evaluate an architecture (e.g. latency), neural systems templates are used. This is possible because the computations performed by neural systems are decomposed hierarchically. They are firstly expressed as high-level functions; e.g. learning one pattern, and are then further divided into low-level computations; e.g. local processing of neurons. Furthermore, neural systems are regular in their processing. Their processing is composed of phases which are clearly defined. These neural computation templates allow the analytical determination of the speedup when implementing each particular mapping scheme.

Neural systems perform generally two high level transforms; a learning (adaptation) phase and a relaxation (recalling) phase. The main activity of the learning phase corresponds to the modification of the connections weights to the presented environment. While, in the relaxation phase the neurons simply propagate their signal in accordance with the weights that are now fixed. Some neural systems predetermined the weights of their connections; for example the Boltzmann machine when used as a combinatorial optimiser [Aar87a]. However, these types of neural systems might still be decomposed along their connections; i.e. the evolution of the neurons always relates to
the weight of their connections. Hence, the job of the analytical framework is to evaluate these two computations, this is done for the different mapping schemes. We, however, concentrate on the learning phase of neural systems because it is the most computationly demanding [Ref90a]. The analytical framework applies similarly to the relaxation phase.

It has been shown how a mapping scheme is parallelising a computational aspect of a neural system computation. For example, the net replication parallelises the highest level transform; i.e. the adaptation and relaxation phases, whilst a neuron mapping parallelises the transforms applied to a particular set of neurons. Hence, the framework identifies the optimal mapping scheme in three stages:

(a) Evaluation of the speedup for the parallelised computational aspect only; referred to as relative speedup,
(b) Incorporation of the relative speedup within the high level transform applied by the neural system. This gives the absolute speedup of the mapping schemes (6.2).
(c) Identification of the optimal mapping scheme according to the ranking based on their absolute speedup.

The following section describes how relative speedups and absolute speedups of neural systems mapping schemes are analytically evaluated, and shows how the optimal mapping is then identified.

6.2.2. Analytical Selection of the Optimal Mapping Scheme

Before the relative speedups and absolute speedups of neural systems mapping schemes are analytically expressed, the following notations are defined.

Notations

- A neural system is represented by an ordered set of training patterns, slabs, neurons and connections, \(\mathcal{N} = (N_p, s, N_1, N_2, \ldots)\) such as:
  - \(N_p\) = total number of training patterns.
  - \(s\) = total number of slabs. A slab is a group of neurons or connections that generally performs the same activation or learning rule.
  - \(N_i\) = number of slabs that can process in parallel. These are groups of neurons or connections that do or do not perform the same rule but that can operate in parallel. Typically they are represented by layers of neurons or connections that interconnect two layers of neurons.
  - \(N_i\) = number of neurons in slab \(i\). A slab of neurons that performs in parallel is generally referred to as a layer in a supervised neural model or a cluster in an
unsupervised neural system.

\( N_{ij} \) = number of connections between slab i and slab j. Generally such connections connect two adjacent layers (or clusters) of neurons.

- A parallel machine is defined relative to the processing occurring in a neural system. Such a virtual/abstract parallel machine is represented by a tuple \( M_s = (p, t_i, O_i) \) such as:

\[ p = \text{number of processing elements } PE_a. \] Typically, the processing elements have multiply, add, and basic load/store operations.

\[ t_i = \text{processing time for the element}_i \text{ on the processor } PE_a. \] The element \( i \) can be a connection, a neuron, a slab or a pattern. \( t_i \in \{ t_c, t_n, t_s, t_p \} \)

\[ st_i = \text{processing time for the element}_i \text{ on sequential machine That is:} \]

\( st_i \in \{ st_c, st_n, st_s, st_p \} \)

\( O_i = \text{overhead time taken by the processing of the connection, neuron, slab, or pattern. Typically such overhead time can be expressed as a function of virtual times } t_{load}, t_{com}, t_{recover} \) where:

\[ O_i = F(t_{load}, t_{com}, t_{recover}) \]

\[ t_{load} = \text{time taken to load data on one processor } PE_a. \]

\[ t_{recover} = \text{time taken to recover data from one processor } PE_a. \]

\[ t_{com} = \text{time taken to exchange data from one processor } PE_a \text{ to another processor } PE_b. \]

The expression of \( O_i \) refers to the processor-network communication \( (t_{load}, t_{recover}) \) as well as to the inter-processor communication \( (t_{com}) \). Hence, the evaluation of \( O_i \) typically depends on the available schemes to load or retrieve data on the whole network of processors or on a "part" of the network of processors. For example, a 2D grid massively parallel machine, e.g. DAP, provides efficient communication mechanisms to load specific rows or columns within the network of processors. Moreover, \( O_i \) depends on how much inter-processor communication is required for the processing of the element \( i \) and also on the machine inter-processor schemes available. Moreover, these overheads depend on the data required by the processing, the current location of these data, and how these data are to be sent. The data required are determined by the neural system, their location depends on the mapping scheme specific of the machine type, and their transfer depends on the communication schemes made available by the machine.
Grain of Mapping

For each basic mapping scheme, the notion of grain of mapping is defined as follows:

- \( g_c \) = grain of the connection mapping.

\[
g_c = 1 \text{ if } \frac{N_c}{p} \leq 1 \text{ otherwise } g_c = \text{int} \left( \frac{N_c}{p} \right)
\]

In fact this means the \( g_c \) is the number of connection(s) that each processor \( PE_a \) should embed. This term is related to the number of connections \( N_c \), that can process in parallel and to the number of processors \( p \).

- \( g_n \) = grain of the neuron mapping.

\[
g_n = 1 \text{ if } \frac{N_l}{p} \leq 1 \text{ otherwise } g_n = \text{int} \left( \frac{N_l}{p} \right)
\]

\( g_n \) is the number of neuron(s) that each processor should embed, and is related to the number of neurons \( N_l \) as well as the number of processors \( p \).

- \( g_s \) = grain of the slab mapping.

\[
g_s = 1 \text{ if } \frac{N_t}{p} \leq 1 \text{ otherwise } g_s = \text{int} \left( \frac{N_t}{p} \right)
\]

\( g_s \) is the number of slab(s) that each processor should embed, and is related to the number of slabs \( N_t \), as well as the number of processors \( p \).

- \( g_t \) = grain of the net replication (training pattern mapping.)

\[
g_t = \text{int} \left( \frac{N_t}{p} \right)
\]

\( g_t \) is the number of net(s) that the whole set of processors can embed, and is related to the number of processors \( p \) and the neural system configuration as well as the capacity of each processor in terms of memory and operations capacity.

The terms \( g_i \) depends upon the number of processors and the number of connections, neurons, slabs, and patterns. The term \( N_i \) qualifies the specific neural system and its application in terms of connectivity (\( N_c \) full or sparse), of topology (\( N_l \) and \( N_t \) ) and in terms of training set size (\( N_t \)). When implementing one of the mapping, \( g_i = 1 \) indicates that the parallel machine has enough processors so as to implement the optimal mapping; i.e. each processor representing only one element (of the distributed set).

Relative Speedup

To quantify the benefit of implementing a mapping scheme, the notion of relative speedup is introduced. The relative speedup relates only to the parallelised computation of a neural system operations; i.e. it relates to the operations involved in the mapping (as defined when describing the alternative decomposition schemes of neural systems).
The relative speedup of a mapping scheme along the element $i$ is the ratio of the time taken by the sequential execution of the elements $i$ over the time taken by their parallel execution.

Formally, the relative speedups of the basic mapping schemes (connection, neuron, slab and net) are:

- For a slab of connections $N_c$, the relative speedup ($N_c$, $s_c$ versus $g_c$, $t_c$) is:
  \[ G_c = \frac{N_c}{g_c} \left(1 + O_c\right) \]  
  \[(6.4a)\]

- For a slab of neurons $N_i$, the relative speedup ($N_i$, $s_n$ versus $g_n$, $t_n$) is:
  \[ G_n = \frac{N_i}{g_n} \left(1 + O_n\right) \]  
  \[(6.4b)\]

- For the slabs $N_s$, the relative speedup ($N_s$, $s_t$ versus $g_t$, $t_t$) is:
  \[ G_s = \frac{N_s}{g_t} \left(1 + O_s\right) \]  
  \[(6.4c)\]

- For the patterns $t_p$ the relative speedup ($N_p$, $s_{tp}$ versus $g_p$, $t_{tp}$) is:
  \[ G_{tp} = \frac{N_p}{g_{tp}} \left(1 + O_{tp}\right) \]  
  \[(6.4d)\]

The speedup measure $G_x$ depends on the neural system as well as on the parallel machine. They comprise two terms, the first term $N_x / g_x$ relates to the neural system's configuration and the machine size. They are computed when the number of processors of the machine and the particular neural system configuration ($N_c$, $N_i$, ...) are known. The second term $O_x$ relates to the overheads associated with each element; connection, neuron, slab and pattern. These are harder to compute; they depend on the basic machine communication time as well as on the scheme used for the inter-processor communication and processor-network loading.

**Absolute Speedup**

In order to be able to select the best mapping scheme when implementing a specific neural system on a targeted parallel machine the different relative speedups $G_i$ have to be evaluated on the same basis. Each of the speedup measure described above is relative to a time unit, made up of real time and overhead time ($t_e$, $O_t$). For a connection mapping $G_c$ is evaluated on the time unit $t_c$, whilst for a neuron mapping $G_n$ is evaluated on the time unit $t_n$ (and so on for the other mappings). The relations between $G_x$ and time unit can mislead the selection of a mapping scheme. For example, if a speedup measure $G_1$ is greater than a speedup measure $G_2$ and if they relate to time units $t_1$, $t_2$ such as $t_1 < t_2$, then it might be more beneficial overall to implement the mapping scheme according to $G_2$. Moreover, each speedup $G_i$ is relative to the nature of the neural system; i.e. it is relative to the series of processing steps that describe the neural system. A specific neural system
defines a particular set of slabs, neurons and connections that compute in parallel. Thus, it defines many $G_i$, each related to a mapping scheme and a neural’s configuration. The processing that a particular neural system performs dictates how many times a relative speedup $G_i$ will be exploited. Hence, to evaluate the overall benefit of each mapping scheme the **absolute speedup** $\Theta_i$ is defined such as:

The **absolute speedup** $\Theta_i$ corresponds to the ratio of the time taken to execute sequentially the whole application over the time taken to execute the whole application when implementing the mapping along the element $i$.

$\Theta_i$ needs the expression of the total time for executing sequentially a neural system ($T_s$); $\Theta_i = T_s / T_i$. This is expressed in a simplistic form as follow:

$$
T_s = \sum_{\text{patterns}} t_{fp}
$$

$$
T_s = \sum_{\text{patterns}} \left[ \sum_i N_i^x t_n + \sum_j N_j^x t_c \right]
$$

where

$N_i^x$ = number of neurons in slab $x$, and $N_j^x$ = number of connections in slab $y$

This is obtained from the following considerations. The total time for a neural system processing is the time spent on processing the whole set of training patterns. Thus, the total time is the sum of the time taken to process each pattern ($2$). The time taken by the processing of a pattern can be expressed as a function of the time taken by the processing of the neurons per slabs plus the time taken by the processing of the connections per slabs. Taking Equation (6.5) the total time taken to execute in parallel a neural system mapping scheme $i$, defined as $T_i$, can now be expressed using the expressions of their relative speedups. This gives:

- For the connection mapping $T_c$ is:
  $$
  T_c = \sum_{\text{patterns}} \left[ \sum_i N_i^x t_n + \sum_{ij} \frac{1}{G_i} t_c \right]
  $$
  (6.6a)

- For the neuron mapping $T_n$ is:
  $$
  T_n = \sum_{\text{patterns}} \left[ \sum_i \frac{1}{G_i} t_n + \sum_j N_j^x t_c \right]
  $$
  (6.6b)

- For the slab mapping $T_s$ is:
  $$
  T_s = \sum_{\text{patterns}} \left[ \frac{N_i^x}{G_i} t_n + \frac{N_j^x}{G_i} t_c \right]
  $$
  (6.6c)

- For the net replication $T_{fp}$ is:
  $$
  T_{fp} = \sum_{\text{patterns}} \left[ \frac{1}{G_{fp}} t_{fp} \right]
  $$
  (6.6d)
These expressions have been derived from Equation 6.5, from the expressions of the relative speedups (6.4a to 6.4d), and from the following considerations.

**Firstly**, the configuration, i.e. number of slabs, number of connections, defines a certain number of relative speedups for each mapping. For example a 3 layer \((N_1, N_2, N_3)\) fully interconnected \((N_1^{32}, N_2^{23})\) Multi Layer Perceptron defines the following relative speedups:

- \(G_{c^{12}}, G_{c^{23}}\) for the connection mapping. That is, a speedup \(G_{c^{12}}\) applies when the connections from the first layer to the second layer are processing in parallel and a speedup \(G_{c^{23}}\) applies when the connections from the second layer to the third layer are operating in parallel. That gives two relative speedups when implementing a connection mapping.

- \(G_{n^{2}}, G_{n^{3}}\) for the neuron mapping. That is, a speedup \(G_{n^{2}}\) applies when the neurons within the second layer operate in parallel and a speedup \(G_{n^{3}}\) when the neurons within the third layer operate in parallel. That gives two relative speedups when implementing a neuron mapping.

**Secondly**, the series of processing for a particular neural system (and model) defines the number of times \(\cup_{r}\) each relative speedup is going to be used. For example, using the previous model, and considering the forward and backward phases, the terms \(G_{c^{12}} \& G_{c^{23}}\) are used two times each during one learning cycle.

### Optimal Mapping Scheme

Each of the \(Ti\) measures represents the overall time taken for executing a neural system when implementing a mapping scheme along the \(\text{element}_i\) (basic mapping scheme). The speedup for each mapping scheme is expressed as the ratio \(\Theta_i = T_i / T_i\) (6.2). The **optimal mapping** is the one that offers the best speedup; i.e. the maximum \(\Theta_i\). The expression of the absolute speedup for each mapping scheme is obtained by replacing in equations (6.6a) to (6.6d) the relative speedups with their original expressions ((6.4a) to (6.4d)). The communication overheads are taken into account within the definition of the relative speedups. Hence, the final equations expressing the absolute speedup to expect when mapping a neural system onto a parallel machine is expressed as a function of the neural system components (e.g. number of slabs, neurons ...) and as a function of the targeted parallel machine (e.g. number of processors, communication cost ...).

The optimal mapping scheme for the implementation of a neural system on a parallel machine is thus identified by evaluating the different \(\Theta_i\). The optimal mapping scheme \(\Theta_{\text{optimal}}\) is such that:

\[
\Theta_{\text{optimal}} = \text{MAX} (\Theta_1, ..., \Theta_i, ..., \Theta_n)
\]

(6.7)

which is equivalent in terms of parallel timing to:
The alternative mapping schemes of a neural system (represented by their expression $T_i$) are thus ranked; and it is the scheme offering the minimal $T_i$ which is chosen for implementation on the parallel machine.

### 6.3. Backpropagation Decomposition Schemes

This section firstly overviews backpropagation neural systems and its computations. Secondly, the alternative decomposition schemes, both basic and multiple are identified following the Abstract Decomposition presented in Chapter 4.

#### 6.3.1. Backpropagation Neural Systems

As described in Chapter 2, the function of backpropagation systems is to retain a set of input-output mappings. It is trained by presenting a series of input-output mappings, the network adjusting itself (via its weighted connections) so as to retain the mapping. Each training cycle consists of three phases: a forward phase where each neuron computes its new state, a backward phase where each neuron computes its error and an update phase where each connection adapts its weight. The complete processing of an application means that each pattern gets fully learned: the three computational phases are applied until the error term meets a particular value referred to as the tolerance parameter. In the following sections we shall assume that the decomposition scheme does not interfere with the convergence dynamics. The processing phases are detailed and formally described, this is done so as to set the notation and prepare the ground for the analytical framework.

- $N_{input}$ = number of neurons in the input layer
- $N_{h}^{i}$ = number of neurons in the hidden layer $i$
- $N_{output}$ = number of neurons in the output layer
- $N_{input}^{i}$ = number of connections from the input or hidden layer ($i$) to the output or hidden ($i+1$) layer
- $N_{tp}$ = number of training patterns
- $N_{h}$ = number of hidden layers

The processing occurring during the training of a backpropagation system corresponds to a series of cycles. Each cycle is formed by a forward phase, a backward phase and finally a weight update phase. The complete sequential processing ($T$), using (4.5), can then expressed as:

$$ T = \sum_{patterns} \left( T_{forward} + T_{backward} + T_{update} \right) $$

In practise, the convergence dynamics can be affected by the decomposition scheme but only to the extend that different machines support different precision arithmetic.
where:

\[ T_{\text{forward}} \]

the overall time spent in the forward phase. During this phase, and starting with the hidden layer, each neuron computes a non-linear transform to its net input to yield to the neuron's output state. All neurons within a same layer can compute in parallel, this is referred to as intra-slab parallelism.

\[ T_{\text{backward}} \]

the overall time spent in the backward phase. During this phase, and starting with the output layer, each neuron computes its error term. The computation of the error term for both output neurons and hidden neurons is described in Chapter 2. This phase also exhibits intra-slab parallelism.

\[ T_{\text{wup}} \]

the overall time spent in the weight update phase. During this phase, each weight updates its value, according to the backpropagation learning rule. All weights can update in parallel, this implements both intra and inter slab parallelism.

The backward computations for the hidden neurons differ from the computations for the output neurons. This fact is used to further decompose (6.9) thus yielding to the following expression (6.10):

\[ T = \sum_{\text{patterns}} \left[ \sum_{\text{out}} \left( N_h^N_p + N_p^\text{wup} \right) t_{\text{forw}} + N_h^N_p t_{\text{backw}} + N_p^\text{wup} t_{\text{backw}} + (N_h^N_p + N_p^\text{wup}) t_{\text{wup}} \right) \]

where:

\[ t_{\text{forw}} \] (\( T_{\text{forw}} = \Sigma_{t_{\text{forw}}} \)) - time taken by a neuron for its forward phase; i.e. computation of its new state,

\[ t_{\text{backw}} \] (\( T_{\text{backw}} = \Sigma_{t_{\text{backw}}} \)) - time taken by a neuron for its backward phase,

\[ t_{\text{wup}} \] (\( T_{\text{wup}} = \Sigma_{t_{\text{wup}}} \)) - time taken by a connection to update its weight.

Because we assume that the decomposition scheme does not interfere with the convergence dynamics, the evaluation of (6.2) is simplified by considering only the terms inside the summation.

6.3.2. Decomposition Schemes

As described in Chapter 4, there exists basic and multiple decomposition schemes of neural systems. Basic decomposition schemes are exploiting a unique type of parallelism, whilst their combination, i.e. multiple decomposition schemes exploit more than one type of parallelism. Table 6.1 lists the basic decomposition schemes of backpropagation systems and indicates their spatial and temporal information. It specifies the elements of backpropagation neural systems that are to be distributed over a
network of processors, and gives the number of these elements as parameters.

<table>
<thead>
<tr>
<th>Decomposition Scheme</th>
<th>Spatial Information i.e. Distributed Element &amp; Number</th>
<th>Temporal Information i.e. Operation Involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection decomposition</td>
<td>( w_{ij} ) &amp; ( N_{c_{ij}} ) &amp; ( N_{output} )</td>
<td>Weight Update Net Input</td>
</tr>
<tr>
<td>neuron decomposition</td>
<td>( o_i ) &amp; ( N_{c_{i}} ) &amp; ( N_{input} )</td>
<td>Forward Backward</td>
</tr>
<tr>
<td>connection-layer decomposition</td>
<td>Layer ( W_{ij} ) &amp; ( N_{hidden_layers} )</td>
<td>Layer Weight Update</td>
</tr>
<tr>
<td>training decomposition</td>
<td>Network ( N_{tp} )</td>
<td>High Level Transform</td>
</tr>
</tbody>
</table>

Table 6.1 - Spatial & Temporal Information for the Basic Decomposition Schemes -
This table presents the spatial and temporal information of the basic decomposition schemes. The spatial information describes the elements (and their number) distributed by each decomposition. The temporal information specifies the operations parallelised by each decomposition.

For each decomposition scheme, the spatial information specifies the elements distributed. They are respectively: the connection weights \( w_{ij} \) for the connection decomposition, the neuron state \( o_i \) and the error term \( \delta_i \) for the neuron decomposition, the layer-connection \( W_{ij} \) for the connection-layer decomposition, and the whole network for the training decomposition. The temporal information specifies the operations parallelised by each decomposition scheme. They are respectively: the weight update and net input computations for the connection decomposition, the forward and backward computations for the neuron decomposition, the layer weight update for the connection-layer decomposition, and the high level transform (i.e. learning of one pattern) for the training decomposition.

Basic decomposition schemes can be combined to form multiple decomposition schemes. In this case, three sets of multiple decomposition schemes are identified: schemes that combine two basic schemes \( C_2=6 \), schemes that combine three basic schemes \( C_3=4 \), and finally the scheme that combines all basic schemes \( C_\Sigma=1 \).

The first set identifies connection and neuron decomposition, connection and training decomposition, neuron and training decomposition. The combination of connection and neuron decomposition exploits the intra-neuron, inter-layer, and inter-layer types of parallelism. In this case, both neurons and connections that belong to a same layer are distributed over a network of processors. The combination of connection and connection-layer decompositions distributes the whole set of connections. Indeed, the connection decomposition distributes connections that belong to a same layer, while connection-layer requires connections from different layers to be distributed. The combination of connection and training decomposition exploits the intra-neuron and training types of parallelism. This scheme implies that each replicated network is distributed along its connections, on a layer basis. Similarly for the combination of
neuron and training decomposition, i.e. each replicated network is itself distributed along its neurons, still on a layer basis. The combination of neuron and connection-layer decompositions defines a scheme where neurons of a same layer are distributed, and connections of different layer are distributed. The combination of connection-layer and training decompositions defines a scheme where each replicated network distributed its layers of connections.

The second set determines four more multiple schemes $C^4$. Firstly, the combination of connection, neuron, and connection-layer which distributes neurons on a layer basis, and all the connections; i.e. connections that belong or not to a same layer. Secondly, the combination of connection, neuron and training decompositions. In this case, each replicated network is itself distributed along its neurons and connections, both on a layer basis. Thirdly, the combination of connection, connection-layer and training decompositions. In this case, each replicated network is itself distributed along the whole set of connections. Finally, the combination of neuron, connection-layer and training decompositions. In this case, each replicated network is itself distributed along its neurons on a layer basis, and along connections of different layers.

Finally, the combination of all basic schemes determines a multiple scheme that exploits all types of parallelism. This scheme implies that each replicated network is distributed along its neurons on a layer basis, and along the whole set of connections.

There are $4 + C^4 + C^3 + 1 = 4 + 6 + 4 + 1 = 15$ decomposition schemes, both basic and multiple, each described in a machine-independent manner. The basic schemes are defined in terms of data layout; i.e. identification of the distributed element(s), and in terms of operations involved in the parallelisation. At this stage, we emphasise the fact that the decomposition does not recommend any particular decomposition scheme, but proposes them to the analytical framework of the Machine Dependent Decomposition. However, it is quite obvious that some of these schemes are not optimal at all; for example the combination of connection-layer and training decompositions. The next section presents the software of targeted parallel machine: the DAP.

6.4. Analytical Speedup Evaluation

This section presents the usage of the analytical framework proposed above, for the implementation of backpropagation systems onto the DAP machine. It specifies the alternative mapping schemes of backpropagation, and shows how the optimal scheme is analytically identified.

This section is divided into four parts. The first part overviews the DAP machine. The second part reviews the expression of both relative and absolute speedups for the basic mapping schemes. The third part specifies the mapping schemes of backpropagation neural systems and their expected speedup. These schemes are now
specified in a machine-dependent manner; i.e. they correspond to the decomposition schemes identified in section 6.2 when implemented onto the DAP. Finally, the fourth part describes how the optimal mapping is identified.

6.4.1. Distributed Array of Processors

The architecture of the Distributed Array of Processors (DAP) has been described in Chapter 2, and its programming language, i.e. Fortran Plus has been overviewed in Chapter 3. This section focuses on the DAP programming environment used for implementing the mapping schemes of backpropagation neural systems.

The DAP is programmed in Fortran Plus which is an extension of the Fortran language designed to support matrix operations. This language allows algorithms to include parallel vector and matrices operations, and to consequently take advantage of their parallelism. An overview of the DAP system is given in the figure below, Figure 6.2.

![Figure 6.2 - DAP System Overview](image)

Typically, a DAP application is composed of a host program which handles the host-based I/O and user interaction, and a DAP program which uses the parallel features of the DAP. The DAP program resides in the code memory of the Master Control Unit, shown in Figure 6.2, and is accessible through a host computer (VAX or Sun). According to the application the control is wholly given to the DAP or the host remains in control; in this case the DAP only runs the highly data parallel functions. Furthermore, the DAP edge size and the program matrices size do not have to fit. That is, the compiler is responsible for the actual decomposition of the matrix so as to fit the edge size. It performs a tiling algorithm that enables programs not to take into account the actual edge size. However, in our case this issue is addressed in the next section which evaluates the different.

---

8 Fortran Plus is a trademark of AMT.
mapping schemes. Data transfer between the host program and the DAP program is done via variables declared as "common". An application executing onto the DAP is thus made of two components: the host program written in C, and the DAP program written in Fortran Plus. An host program typically consists of the five phases: opening the connection to the DAP (dapcon_), transferring data to the DAP (dapsen_), giving control to the DAP (dapent_), retrieving data (daprec_) and regaining control from the DAP (daprel_). A DAP program typically consists of four phases: convert the data received from the host (convhtod), execute, convert data from the DAP format to the host (convdtoh), and release the DAP (return).

Fortran Plus is a matrix-vector oriented language; a program corresponds to a set of matrix-matrix or matrix-vector operations. Parallelism is denoted as follows:

```fortran
real amatrix(*64,*64)
......
amatrix = 2 * amatrix
```

Figure 6.3 - Parallelism as Denoted in Fortran Plus -
A matrix 'amatrix' of dimension 64 by 64 is declared as Parallel; each element of this matrix is then multiplied by a factor 2, simultaneously for all elements of the matrix.

As indicated in Chapter 3, Fortran Plus provides indexing techniques to select elements from matrices and/or vectors. For example, `amatrix(l,)` selects the first row of the matrix "amatrix" as declared in Figure 6.3. Matrices construction functions are also provided, they allow the construction of matrix from vectors and/or scalars, and/or matrix. For example, `MATC(v,5)` constructs a matrix where each column is equal to the vector argument v. A complete description of the Fortran Plus syntax and associated matrix/vector construction tool is can be found in [AMT90a].

6.4.2. Relative & Absolute Speedup

Notations

The notations have already been defined in section 6.2.2, therefore only the expressions describing backpropagation neural systems and the DAP are given below.

- A backpropagation neural system is represented by an ordered set of training patterns, slabs, neurons and connections, \( NN_i = (N_p^s, N_i^d, N_{c1}, N_{c2}, N_{c3}, \ldots) \).
- The DAP machine is represented relatively to the processing occurring in backpropagation neural systems by the tuple \( M = (p, t_i, O_i) \) such as:
Grain

The **grain of the decomposition** for each basic scheme is as follows:

- $g^\beta$ = grain of the connection decomposition. ($g^\beta = 1$ if $\frac{N^\beta}{p} \leq 1$, otherwise $g^\beta = \text{int}(\frac{N^\beta}{p})$).

- $g^\gamma$ = grain of the neuron decomposition. ($g^\gamma = 1$ if $\frac{N^\gamma}{p} \leq 1$, otherwise $g^\gamma = \text{int}(\frac{N^\gamma}{p})$).

- $g^l$ = grain of the slab decomposition. ($g^l = 1$ if $\frac{N^l}{p} \leq 1$, otherwise $g^l = \text{int}(\frac{N^l}{p})$).

- $g_r$ = grain of the net replication (number of net(s) the set of processors embeds).

**Relative Speedup**

For the backpropagation neural systems, the relative speedup terms of the four basic mapping schemes are as follows:

- For a slab of connections $N^\beta$ the relative speedup ($N^\beta st_c$ versus $g^\beta t_c$) is:

  $$G_{c^\beta} = N^\beta g^\beta (1 + O_c)$$  \hspace{1cm} (6.11a)

- For a slab of neurons $N^\gamma$ the relative speedup ($N^\gamma st_n$ versus $g^\gamma t_n$) is:

  $$G_{n^\gamma} = N^\gamma g^\gamma (1 + O_n)$$  \hspace{1cm} (6.11b)

- For the slabs $N^l$ the relative speedup ($N^l st_s$ versus $g^l t_s$) is:

  $$G_{s^l} = N^l g^l (1 + O_s)$$  \hspace{1cm} (6.11c)

- For the patterns $t_p$ the relative speedup ($N^p st_p$ versus $g^p t_p$) is:

  $$G_{p^p} = N^p g^p (1 + O_p)$$  \hspace{1cm} (6.11d)

**Absolute Speedup**

The absolute speedup for a mapping scheme $i$ is defined such that:

$$\Theta_i = \frac{T}{T_i}$$  \hspace{1cm} (6.12)

$$T = \Sigma_{\text{patterns}} t_p$$

$$T = \Sigma_{\text{patterns}} \left[ \bigcup_j N^p_j t_n + \bigcup_j N^c_j t_c \right]$$  \hspace{1cm} (6.13)

$T =$ The total time for executing **sequentially** a neural network

$T_i =$ The total time taken for the execution of the mapping scheme $i$.

The absolute speedups for each basic schemes are as follows:

- For the connection mapping: $\Theta_c = \frac{T}{T_c} ; \quad T_c = \Sigma_{\text{patterns}} \left[ \bigcup_i N^c_i t_n + \bigcup_j \frac{1}{O^c_c} t_c \right]$  \hspace{1cm} (6.14a)
• For the neuron mapping: $\Theta_n = T/T_n$ ; $T_n = \Sigma_{\text{patterns}} \left[ \frac{1}{G_n} t_n + \sum_j N^{\ell_j} t_c^j \right]$ (6.14b)

• For the slab mapping: $\Theta_s = T/T_s$ ; $T_s = \Sigma_{\text{patterns}} \left[ \frac{N_J}{G_s} t_n + \frac{N_J}{G_s} t_c^s \right]$ (6.14c)

• For the net replication $T_{ip}$ is: $\Theta_{ip} = T/T_{ip}$ ; $T_{ip} = \Sigma_{\text{patterns}} \left[ \frac{-1}{G_{ip}} t_{ip} \right]$ (6.14d)

Having reviewed the absolute speedup for the four basic mapping schemes, the speedups of both basic and multiple mapping schemes for backpropagation systems onto the DAP are now detailed.

6.4.3. Mapping Schemes

The set of decomposition schemes determined earlier is evaluated along with the features of the DAP machine. The implementation of these decomposition schemes, in other words the set of mapping schemes, is reduced before any attempt to forecast their speedups is made. This is done because of the characteristics of the DAP machine and its programming environment; i.e. the Fortran Plus language. The language is dedicated to vectors and matrices, hence the mapping schemes are expressed as sets of vector and/or matrix operations. The expression of backpropagation systems in a matrix algebra form specifies layers of neurons as vectors, and layers of connections as matrices. Expressing backpropagation systems in matrix-algebra corresponds to the implementation of the neuron and connection multiple decomposition scheme (as described by the AD module). The combination of neuron and connection mapping is thus the primitive mapping scheme. This scheme offers $\Theta_{c+\Sigma} = T/T_{c+\Sigma}$ for absolute speedup.

Furthermore, bearing in mind that the number of connections in a slab (that can operate in parallel) is always greater than the number of slabs (that can operate in parallel), and according to the expressions (6.11a) to (6.11d) one obtains:

$$N_J > N_f \land N_J > N_f \land N_J > N_f \text{ which implies } \Theta_{c+\Sigma} > \Theta_c \land \Theta_{c+\Sigma} > \Theta_n \land \Theta_{c+\Sigma} > \Theta_i$$

According to this, the connection mapping $\Theta_c$, the neuron mapping $\Theta_n$, and the connection-layer $\Theta$, can be disregarded because always offering lower speedups. Therefore, four mapping schemes are available for evaluation:

- the combination of neuron & connection mapping,
- the net replication mapping,
- the combination of neuron & connection & net replication mapping,
- the combination of neuron & connection & connection-layer & net replication mapping.

Table 6.2 presents each of these mapping schemes, their respective scheme onto the DAP, and the expression of their absolute speedup.
Table 6.2 - The Set of Mapping Schemes to be Evaluated

<table>
<thead>
<tr>
<th>Mapping Scheme</th>
<th>Corresponding Mapping Scheme onto the DAP</th>
<th>Absolute Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>combination of connection &amp; neuron</td>
<td>matrix-vector</td>
<td>$\Theta_{c+n} = T/T_{c+n}$</td>
</tr>
<tr>
<td>net replication</td>
<td>training</td>
<td>$\Theta_{ip} = T/T_{ip}$</td>
</tr>
<tr>
<td>combination of connection &amp; neuron &amp;</td>
<td>matrix-matrix</td>
<td>$\Theta_{c+n+ip} = T/T_{c+n+ip}$</td>
</tr>
<tr>
<td>net replication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>combination of connection &amp; neuron &amp;</td>
<td>distributed matrix-matrix</td>
<td>$\Theta_{c+n+ip} = T/T_{c+n+ip}$</td>
</tr>
<tr>
<td>connection-layer &amp; net replication</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first column, entitled Mapping Scheme in the Table, presents each mapping scheme as the combination (or not) of basic schemes. The second column presents the name each mapping scheme is referred to within the DAP environment. This is done in order to relate the naming of mapping schemes with their actual implementation. Finally, the last column lists the expression of the absolute speedups. Each absolute speedup is defined as a combination of the absolute speedup of the basic schemes expressed earlier in equations (6.14a) to (6.14d). For example, $\Theta_{c+n}$ combines $\Theta_c$ for the basic connection mapping, and $\Theta_n$ for the basic neuron mapping.

As shown in Table 6.2, the matrix-vector mapping corresponds to the combination of neuron and connection mapping schemes. Both connections and neurons (belonging to the same layer) are distributed across the grid of processors. The training mapping defines a scheme where each processor represents a complete backpropagation. The matrix-matrix mapping corresponds to the combination of connection, neuron and training mappings and defines a scheme where each replicated backpropagation implements a neuron and connection mapping. In this case, each replicated network implements a matrix-vector mapping. Finally, the distributed matrix-vector mapping implements a combination of connection, neuron, connection-layer, and training mappings, and defines a scheme where each replicated backpropagation is itself distributed over a part of the processors' network. Each replicated network implements a neuron, connection, and connection-layer mappings.

Each mapping scheme is now detailed, their execution timing is expressed relative to their sequential timing expression (6.10), and to their relative speedups (6.11a) to (6.11d). This enables the alternative mapping schemes to be ranked, and hence to be analytically compared to retrieve the optimal scheme.

Matrix-Vector Mapping

Taking activities values of neurons as vectors and the set of connection weights as matrices, the network is represented as a series of matrix-vector multiplications and non-linear transforms. Figure 6.4 illustrates the matrix-vector mapping scheme as expressed as a set of matrix-vector operations.
Figure 6.4 - Matrix-Vector Mapping Scheme -

$V_i$ represents the vectors of neurons in layer (i). $W$ represents the synaptic matrix between layer (i) and layer (i+1). Each element of this matrix; $w_{ij}$ corresponds to the weight value of the connection linking neuron (j) of layer (i) to neuron (i) of layer (i+1).

This mapping exploits the first two types of parallelism: the *intra-neuron* and *intra-layer*. The intra-neuron parallelism is exploited when a neuron computes its net input; i.e. each individual multiplication occurs concurrently. Moreover, to sum the individual products, the recursive doubling technique is normally used. The intra-layer parallelism is exploited as follows: each neuron in a layer computes its new activation state (i.e. in parallel with the others); moreover, each connection within the same layer operates its learning rule in parallel. This mapping scheme involves a connection mapping and a neuron mapping, thus yielding to two relative speedup; $G_c$ as defined in (6.11a), and $G_n$ as defined in (6.11b).

Network and machine sizes are taken into account when calculating the storage requirements of vectors and matrices. Vectors are split into $p$-element subvectors, where $p$ is the machine's edge size, and the weight matrices are divided into $p^2$ blocks. The expected speedup depends upon the ratio of the machine size and network size. The size of the weight matrix, otherwise known as the synaptic matrix, is particularly crucial. If the number of neurons per layer does indeed fit the machine, i.e. the neurons’ vector is split into $a_1$ vectors of dimension $p$ (where $p$ is the machine edge size; $p^2$ being the number of processors); then, during the forward and backward phases all neurons compute in parallel. The weight matrices often require in fact $a_2$ times the machine dimension; that is, $W > p^2$ where $W$ is the maximum number of weights between adjacent layers. This increases the execution time by a factor $a_2$. However, in case of sparsely connected networks this mapping ought to adapt and to use the sparse matrix techniques developed for other applications [Gup90a]. From (6.14a), (6.14b), and (6.10), the parallel execution $T_{c+} = \frac{T}{\Theta_{c+}}$ time comes down to Equation 6.15.

Equation (6.15):

$$T_{c+} = \Sigma_{patterns} N_P \left[ \Sigma_{collection} \left\{ \left( \frac{1}{G_{output}} + \frac{1}{G_{hidden}} \right) t_{forw} + \frac{1}{G_{output}} t_{backw} + \frac{1}{G_{output}} t_{backw} + \frac{1}{G_{output}} \right\} + \left( \frac{1}{G_{hidden}} + \frac{1}{G_{output}} \right) t_{back} \right] \right] \tag{6.15}$$
For the sake of simplicity this expression does not take into account the reduction in time occurring during the computation of dot products, where the embedded individual multiplications (i.e. $\Sigma w_i x_j$) occur in parallel; in the terms $t_{\text{forw}}$, and $t_{\text{hidden}}$.

**Training Mapping**

The training mapping replicates the entire network of neurons (neurons that belong to the input, hiddens and output layers) along with a single training pattern in each processor. This requires the processor local memory to be able to hold the whole network of neurons. The host retains the connection weights and broadcasts them along the instruction path. Thus, each processor accumulates simultaneously the activity of a given neuron (in the next layer) in parallel for all the training patterns. Each neuron is then added upon in turn; and the non-linear transform $^9$ is applied to each neuron. This process is repeated for all layers until the output layer. The backward phase is treated in the same way. Finally, to update the weights, we need to accumulate $\Delta w$; i.e. the amount by which each connection weight is to be modified (set of summations over all the processors), note that this is the only time communication is required. This scheme involves a relative gain $G_\alpha$ as defined in (6.11d).

The expected speedup depends upon the ratio of the machine size and training set size; bearing in mind that the processor ought to retain the whole set of neurons. In the case of the DAP, each processor has up to 1Mb of local memory, thus what could have been a restriction becomes an advantage. Ideally$^{10}$, from (6.14d), the execution time is reduced by a factor $G_\alpha$, this gives:

$$T_{wp} = \Sigma_{\text{patterns}} \frac{1}{G_{\text{ip}}} t_{wp}$$

(6.16)

There are two limitative aspects. Firstly, there is no theoretical proof as yet that the learning procedure converges when the weights are updated for a whole set of patterns (as oppose to being updated after each pattern presentation). But experimental results are consistent with this premise [Aze92a]. Secondly, the training set has to be large enough so as to keep the processors busy. In conclusion, the benefit of such a mapping is related to the ratio between training set size and the machine size.

$^9 y = \frac{1}{1 + e^{-\text{net input}}}$

$^{10}$ Overheads; i.e. time to collect the weights changes, are included in the expression of $G_\alpha$. 

119
Matrix-matrix Mapping

This mapping integrates the matrix-vector mapping and training set mapping. This scheme involves connection, neuron mapping within net replication; thus yielding to the relative gains: \( G_c, G_n \) as for the matrix-vector mapping, and \( G_{ip} \) as for the training set mapping. However, these relative gains \( G_i \) are **not equivalent** to the previous one (6.15) and (6.16). They do not exhibit the same grain of mapping; it depends on how many network copies the machine contains. \(^1\) As shown in Figure 6.5, the network of neurons is replicated within the machine processors but each copy is also distributed over a column (or a row) of processors. This is possible because there are usually more processors than the width of the network (maximum number of neurons in a layer).

![Figure 6.5 - Matrix-matrix Mapping Scheme -](image)

**Figure 6.5 - Matrix-matrix Mapping Scheme**

This scheme replicates a complete backpropagation neural system over a column (or row) of processors. This implements a training set mapping along rows of processors; denoted 'training set' in the Figure. Then, each replicated neural system is distributed over the column of processors to implement a neuron & connection mapping, denoted neuron-connection in the figure.

For a network of a width \( n \) and a grid of processors \( p \times p \) (we suppose \( p > n \)) all processors within a row hold a copy of the net, making thus \( p \) copies. \(^2\) Each network copy is spread over a column of processors and therefore exploits both the intra-neuron and inter-layer parallelism during its forward, backward, and weight update phases. Moreover, in the weight update phase after each copy has computed its weight changes (still implementing parallelism) they are summed for the different copies, and thus different training patterns. An obvious disadvantage of this mapping is its complexity in terms of programming.

\(^1\) E.g. 100 processors, 100 neurons gives a grain of 1 for a matrix-vector mapping, but gives a grain of 2 if the machine contains 2 copies of the network, each working with 50 processors.

\(^2\) The waste in processors; i.e. \( n \neq p \) is to be compared with the overall gain.
By integrating the matrix-vector mapping (6.15), and training-vector (6.16) mapping, the execution time reduces to:

\[
= \frac{1}{G^\text{fp}} \sum_{n} \left( \left( \frac{1}{G^\text{hidden}} + \frac{1}{G^\text{output}} \right) t_{\text{forward}} + \frac{1}{G^\text{hidden}} t_{\text{backward}} + \frac{1}{G^\text{output}} t_{\text{backward}} + \left( \frac{1}{G^\text{hidden}} + \frac{1}{G^\text{output}} \right) t_{\text{update}} \right)
\] (6.17)

### Distributed Matrix-vector Mapping

The matrix-vector is further improved by exploiting the inter-layer type of parallelism. According to the number of neurons and the number of processors, it may be worth distributing the neurons and connections that belong to different layers onto different processors. As in the matrix-vector mapping, during the propagation phase the multiplications (whose sum forms the network input) are performed in parallel and each neuron, within the same layer, can then compute its new state in parallel. It is during the update phase that inter-layer parallelism is added; i.e. the weights that connect into the hidden layers and the weights that connect into the output layer are updated in parallel. It is however unlikely that the number of processors will allow such a mapping. That is, the neural network applications are large scale applications, thus the number of connections will outnumber the number of processors. This mapping is thus not adapted to the backpropagation system. However, it might be adapted for other types of neural networks, in particular for the heterogeneous models [Bot91a].

### 6.4.4. Analytical Selection

The selection of the optimal mapping scheme for backpropagation neural systems onto the DAP operates on the following set of mapping schemes:

- Matrix-vector \( \Theta_{c+n} \) described by (6.15),
- Training set \( \Theta_{p} \) described by (6.16),
- Matrix-matrix \( \Theta_{c+n+ip} \) described by (6.17).

The optimal mapping scheme \( \Theta_{\text{optimal}} \), is such that:

\[
\Theta_{\text{optimal}} = \max ( \Theta_{c+n} , \Theta_{p} , \Theta_{c+n+ip} )
\] (6.18)

which is equivalent in terms of parallel timing to:

\[
T_{\text{optimal}} = \min ( T_{c+n} , T_{p} , T_{c+n+ip} )
\] (6.19)

It is always the case:

\[
g^\text{hidden}_{a} \geq g^\text{hidden}_{c} \quad \& \quad g^\text{hidden}_{c} \geq g^\text{hidden}_{c}
\quad \& \quad g^\text{output}_{a} \geq g^\text{output}_{c}
\quad \& \quad g^\text{output}_{a} \geq g^\text{output}_{c}
\]

which implies \( G^\text{hidden}_{a} \leq G^\text{hidden}_{c} \quad \& \quad G^\text{hidden}_{c} \leq G^\text{hidden}_{c}
\quad \& \quad G^\text{output}_{a} \leq G^\text{output}_{c}
\quad \& \quad G^\text{output}_{a} \leq G^\text{output}_{c}
\]
This gives:
\[ T_{c+n} = \sum_{\text{patterns}} N_{tp} \sum_{\text{tol}} A \quad \& \quad T_{c+n+tp} = \sum_{\text{patterns}} \frac{N_{tp}}{G_{tp}} \sum_{\text{tol}} A' \quad \text{with} \quad A \leq A' \] (6.20)

**Firstly**, the selection between the matrix-matrix \( T_{c+n} \) and the matrix-vector \( T_{c+n+tp} \) depends on their respective grain of decomposition; \( g_n \& g_c \) for the matrix-vector mapping, and \( g^* \& g^* \) for the matrix-matrix mapping. The matrix-matrix scheme is necessarily optimal when \( g_n = g_n^* \) and/or \( g_c = g_c^* \). Indeed, in such cases the execution time \( T_{c+n+tp} \) is reduced by a factor \( G_{tp} \) compare to the execution time \( T_{c+n} \).

**Secondly**, the matrix-vector scheme becomes optimal if \( g_n \gg g_n^* \) and/or \( g_c \gg g_c^* \). This is true if the speedup in the expression \( A \) overcomes the speedup due to the factor \( G_{tp} \). We have:

if \( g_n \gg g_n^* \) and/or \( g_c \gg g_c^* \) which implies \( G_n \ll G_n^* \) and/or \( G_c \ll G_c^* \) which in turn implies \( A \ll A' \)

According to (6.20), the matrix-matrix has a relative speedup over the matrix-vector of \( \frac{N_{tp}}{G_{tp}} \), where \( G_{tp} = 1/g_{tp} + g_{tp}O_{tp} \). Hence, \( g_{tp} \) associated with the overheads \( O_{tp} \) determines if the matrix-matrix is optimal. Moreover, \( g_{tp} \) determines the number of backpropagation neural system the machine embeds. Furthermore, when \( g_{tp} \) increases then \( N_{tp}/g_{tp} \) decreases and \( g_{tp}O_{tp} \) increases.

**Thirdly**, the training set is optimal in case of large training sizes. Indeed, in case of small training set sizes compare to network sizes; i.e. \( N_{tp} \ll N_{tp}^l \) & \( N_{tp}^l \), then \( G_{tp} \) increases; that is \( g_{tp} \) (the number of network contained in the machine) decreases. Thus, in case of big training set size, the training set mapping becomes optimal. However, the training set mapping replicates the net over the network of processors, while the matrix-matrix replicates the net and then decomposes each copy across a part of the network of processors. It is always the case:

\( g_{tp} \leq g_{tp}^* \) which implies \( G_{tp} \geq G_{tp}^* \) such that \( T_{c+n+tp} = \sum_{\text{patterns}} \frac{1}{G_{tp}} \sum_{\text{tol}} A' \) & \( T_{tp} = \sum_{\text{patterns}} \frac{1}{G_{tp}} t_{tp} \)

\( t_{tp} > \sum_{\text{tol}} A' \)

So, depending on the overheads \( O_{tp} \) and on the training set size \( N_{tp} \), the training set scheme \( \Theta_{tp} \) is optimal.

**Lastly**, it shall be noted that the matrix-matrix scheme can vary. This scheme involves a connection and neuron mapping within a net replication mapping. As such, it varies according to the number of times the net is replicated. However, it is always the case

\( g_{tp}^2 > g_{tp}^1 \) implies \( g_{tp}^2 \geq g_{tp} \)
Therefore, according to the grain of the net replication $g_p$, and the grain of connection $g_c$, there exists an optimal matrix-matrix mapping. The optimal matrix-matrix is such that:

$$g^1_p < g^2_p \quad \& \quad g^1_c < g^2_c$$

Table 6.3 summarises the optimal mapping scheme and the associated analytical criteria that must be verified.

<table>
<thead>
<tr>
<th>Optimal Mapping Scheme</th>
<th>Analytical Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix-vector</td>
<td>$g^p \gg g^p$</td>
</tr>
<tr>
<td>matrix-matrix</td>
<td>$g^c \geq g^c$</td>
</tr>
<tr>
<td>versus matrix-matrix 1</td>
<td></td>
</tr>
<tr>
<td>versus matrix-matrix 2</td>
<td>$g^1_p &lt; g^2_p \quad &amp; \quad g^1_c &lt; g^2_c$</td>
</tr>
<tr>
<td>training set</td>
<td>$G^p &gt; G^p$</td>
</tr>
</tbody>
</table>

Table 6.3 - Optimal Mapping Scheme & Criteria -

In summary, the matrix-vector mapping scheme is optimal if the grain of decomposition for the matrix-matrix mapping is relatively larger than the one of the matrix-vector mapping. Otherwise, it is the matrix-matrix mapping which is optimal. Moreover, from the variations of matrix-matrix schemes, the optimal scheme is the one with a smaller grain for net replication and smaller grain for connection mapping. Lastly, the training set mapping is optimal when its relative gain is greater than the one of the matrix-matrix mapping scheme.

6.5. Experimental Results

This section presents the experiments conducted in order to validate the analytical predictions. It describes the implementation and execution timing of the matrix-vector mapping, of the matrix-matrix mapping (with different grains) of a backpropagation system. It then shows that the results obtained agree with their analytical predictions.

The backpropagation neural system implemented is a standard one hidden layer network, fully interconnected. The application is a simple digit recognition application, such that the input and output layers have the same number of neurons. The implementation involves several network configurations. In these cases the sizes of the ten digit images, each represented as an array of pixels, are changed accordingly (in order to fit the size of the input layer). The training mapping is ruled out because of the ratio between machine size (64*64) and training set size (10 patterns); as mentioned in the previous section, it is the matrix-matrix and matrix-vector mappings that are implemented onto the DAP. Furthermore, two different matrix-matrix mappings were implemented: the first one established two copies of the backpropagation, and the second one established three copies. The grain of decomposition $g_p$; number of networks the machine holds copies of, are thus respectively 2 and 3. The network considered is $(N_x,N_y,N_z)$ where $N_x$ is the number of neurons in the input and output layers, and $N_z$ the number of neurons in the hidden layer. This gives the same number of connections for
both the input to hidden connection layer and the hidden to output connection layer $N_h$.

According to (6.10), and (6.15) (6.17) the expressions for the matrix-vector timing $T_{c+n}$, the 2 copies matrix-matrix timing $T_{c+n+ip}$, the 3 copies matrix-matrix timing $T_{c+n+ip}$, and the sequential timing $T$ are:

$$T_{c+n} = \sum_{\text{patterns}} \cdot N_p \cdot \left[ \Sigma_{\text{mol}} \left( g^n_h(1+O_n) + g^n_a(1+O_n) \right) t_{\text{forw}} + g^n_h(1+O_n) t_{\text{backw}} + 2g_c(1+O_c) t_{\text{wup}} \right]$$

$$T_{c+n+ip} = \sum_{\text{patterns}} \cdot N_p \cdot \left[ \Sigma_{\text{mol}} \left( g^n_h(1+O_n) + g^n_a(1+O_n) \right) t_{\text{forw}} + g^n_h(1+O_n) t_{\text{backw}} + g^n_a(1+O_n) t_{\text{backw}} + 2g_c(1+O_c) t_{\text{wup}} \right]$$

$$T_{c+n+ip} = \sum_{\text{patterns}} \cdot N_p \cdot \left[ \Sigma_{\text{mol}} \left( g^n_h(1+O_n) + g^n_a(1+O_n) \right) t_{\text{forw}} + g^n_h(1+O_n) t_{\text{backw}} + g^n_a(1+O_n) t_{\text{backw}} + 2g_c(1+O_c) t_{\text{wup}} \right]$$

$$T = \sum_{\text{patterns}} \cdot N_p \cdot \left[ (N_{\text{hidden}} + N_{\text{output}}) t_{\text{forw}} + N_{\text{hidden}} t_{\text{backw}} + N_{\text{output}} t_{\text{backw}} + 2N_{\text{hidden}} t_{\text{output}} t_{\text{wup}} \right]$$

The absolute speedup terms, $\Theta_{\text{mat-mat}}, \Theta_{\text{mat-vec}}, \Theta_{\text{mat-vec}}$, are, respectively such that:

$\Theta_{c+n} = T / T_{c+n}$, and $\Theta_{c+n+ip} = T / T_{c+n+ip}$, $\Theta_{c+n+ip} = T / T_{c+n+ip}$.

The matrix-matrix and matrix-vector mapping schemes are implemented onto the DAP in Fortran Plus, their code can be found respectively in Appendix A and B. Several system configurations are tried: 64-32-64 and 96-32-96 and 96-50-96. These configurations are chosen in order to obtain variations of the grains of decomposition and thus assess the analytical criteria for selection of the optimal scheme. The grains of decomposition for each mapping scheme are as follows:

**64-32-64 configuration**

The matrix-vector mapping defines $g_c=1, g_a=1$, The matrix-matrix with 2 copies of the neural system ($g_p=2$) defines $g_c=1, g_a=1$, The matrix-matrix with 3 copies of the neural system ($g_p=3$) defines $g_c=2, g_a=1$, 

**96-32-96 configuration**

the matrix-vector mapping defines $g_c=1, g_a=1$, The matrix-matrix with 2 copies of the neural system defines $g_c=2, g_a=1$, The matrix-matrix with 3 copies of the neural system defines $g_c=2, g_a=1$, 

**96-50-96 configuration**

The matrix-vector mapping defines $g_c=2, g_a=1$, The matrix-matrix mapping with 2 copies of the neural system defines $g_c=3, g_a=1$, The matrix-matrix with 3 copies of the neural system defines $g_c=4, g_a=1$,

The tables below summarises the results obtained. Firstly, Table 6.4 lists the execution timing of each mapping scheme, and indicates the optimal mapping scheme. This is done for the different configurations of the system. Then, Table 6.5 to Table 6.7

---

13 For simplicity, we assume that the neuron and connection computation times onto the DAP or onto a sequential machine are comparable.
detail the results for each mapping scheme.

<table>
<thead>
<tr>
<th>Neural System</th>
<th>Mapping Scheme</th>
<th>Timing</th>
<th>Optimal Mapping Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-32-64</td>
<td>matrix-vector</td>
<td>148.41</td>
<td></td>
</tr>
<tr>
<td></td>
<td>matrix-matrix</td>
<td>90.035</td>
<td>Matrix-Matrix</td>
</tr>
<tr>
<td></td>
<td>2 copies</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>matrix-matrix</td>
<td>135.34</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 copies</td>
<td></td>
<td></td>
</tr>
<tr>
<td>96-32-96</td>
<td>matrix-vector</td>
<td>175.05</td>
<td></td>
</tr>
<tr>
<td></td>
<td>matrix-matrix</td>
<td>176.35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 copies</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>matrix-matrix</td>
<td>162.719</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 copies</td>
<td></td>
<td>Matrix-Matrix</td>
</tr>
<tr>
<td>96-32-96</td>
<td>matrix-vector</td>
<td>107.387</td>
<td>Matrix-Vector</td>
</tr>
<tr>
<td></td>
<td>matrix-matrix</td>
<td>144.40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 copies</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>matrix-matrix</td>
<td>155.540</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 copies</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4 - Implementation Results -
The matrix-vector scheme is optimal when its grain of decomposition (connections) is relatively smaller compared to the matrix-matrix's grain. Otherwise, it is the matrix-matrix which is optimal. The matrix-matrix with a smaller grain of decomposition for the neuron mapping is optimal when the grain of decomposition for the net replication, i.e. \( g_p < g_p' \), is optimal when the grain of decomposition for the neuron mapping is smaller.

Table 6.5 presents the optimal mapping scheme for different mapping schemes, and configurations of a typical backpropagation neural system. The optimal scheme obtained according to these timings corresponds to the one predicted by the analytical framework. That is, the matrix-vector mapping is optimal when \( g' \gg g \), and the optimal matrix-matrix is such that: \( g_p < g_p' \) & \( g_p^2 < g_p^2' \). That is, \( 2 < 3 \) & \( 1 < 2 \), thus the matrix-matrix with 2 copies is optimal. Otherwise the matrix-matrix with 3 copies is optimal.

Experiments have been carried out in order to assess the framework in more detail. The learning phase of the backpropagation neural system has been executed and timed for the different mapping schemes and for different values of the tolerance parameter. This parameter influences the number of learning passes necessary for the completion of the learning task. Tables 6.5 to 6.7 present the results for a tolerance parameter set to 0.1, 0.05 and 0.01 respectively.
<table>
<thead>
<tr>
<th>Mapping Scheme</th>
<th>Tolerance</th>
<th>Number of Iterations</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-Vector</td>
<td>0.1</td>
<td>20</td>
<td>3.09644</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>33</td>
<td>8.44734</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>87</td>
<td>148.4173</td>
</tr>
<tr>
<td>Matrix-Matrix</td>
<td>0.1</td>
<td>28</td>
<td>2.1039</td>
</tr>
<tr>
<td>2 Copies</td>
<td>0.05</td>
<td>39</td>
<td>5.1811</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>41</td>
<td>90.0351</td>
</tr>
<tr>
<td>Matrix-Matrix</td>
<td>0.1</td>
<td>34</td>
<td>3.26894</td>
</tr>
<tr>
<td>3 copies</td>
<td>0.05</td>
<td>39</td>
<td>8.32735</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>78</td>
<td>135.3458</td>
</tr>
</tbody>
</table>

Table 6.5 - Performances for a 64-32-64 Backpropagation Neural System -
This table presents the execution timing obtained for the three mapping schemes with a 64-32-64 configuration. Each mapping scheme is executed for a tolerance parameter set to 0.1, 0.05, and 0.01. Furthermore, the number of iterations ($\Sigma_{\text{it}}$) required for each mapping scheme is also given. For any value of the tolerance parameter, it is as expected the matrix-matrix which is optimal.

<table>
<thead>
<tr>
<th>Mapping Scheme</th>
<th>Tolerance</th>
<th>Number of Iterations</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-Vector</td>
<td>0.1</td>
<td>23</td>
<td>3.1521</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>25</td>
<td>8.1886</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>39</td>
<td>175.05</td>
</tr>
<tr>
<td>Matrix-Matrix</td>
<td>0.1</td>
<td>27</td>
<td>3.10478</td>
</tr>
<tr>
<td>2 Copies</td>
<td>0.05</td>
<td>35</td>
<td>8.64787</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>45</td>
<td>176.35</td>
</tr>
<tr>
<td>Matrix-Matrix</td>
<td>0.1</td>
<td>22</td>
<td>3.2852</td>
</tr>
<tr>
<td>3 copies</td>
<td>0.05</td>
<td>25</td>
<td>8.3664</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>37</td>
<td>162.7179</td>
</tr>
</tbody>
</table>

Table 6.6 - Performances for a 96-32-96 Backpropagation Neural System -
This table presents the execution timing obtained for the three mapping schemes with a 96-32-96 configuration. Each mapping scheme is executed for a tolerance parameter set to 0.1, 0.05, and 0.01. Furthermore, the number of iterations ($\Sigma_{\text{it}}$) required for each mapping scheme is also given. For any value of the tolerance parameter, it is as expected the matrix-matrix which is optimal.
Table 6.7 - Performances for a 96-50-96 Backpropagation Neural System -
This table presents the execution timing obtained for the three mapping schemes with a 96-50-96 configuration. Each mapping scheme is executed for a tolerance parameter set to 0.1, 0.05, and 0.01. Furthermore, the number of iterations (Σ_n) required for each mapping scheme is also given. For any value of the tolerance parameter, it is as expected the matrix-vector which is optimal.

These results follow what the analytical study predicted. That is, the matrix-matrix mapping scheme is the optimal scheme until the network size reaches a limit, at which time it is the matrix-vector scheme that becomes optimal. It shall be noted that a similar analytical study for the recalling phase can be performed in exactly the same manner just by deleting the terms associated with the phase of weight update $T_{wup}$ in all the expressions (6.14), (6.20) and so on.

6.6. Summary

This chapter developed an analytical framework able to quantify the speedup of alternative neural systems' mapping schemes relatively to the targeted parallel machine. This additional tool is integrated within the Machine Dependent Decomposition module of the System proposed in Chapter 4. The System Abstract Decomposition module provides the set of decomposition schemes according to the types of parallelism available in the neural system. The MDD then analyses the decomposition schemes in relation with the features of targeted machine. With the analytical framework, the MDD module is now able to analytically rank the mapping schemes of a neural system and select the optimal one for implementation.

This chapter has also implemented alternative mapping schemes of backpropagation neural systems onto the Distributed Array of Processors (DAP), and has shown that the experimental speedups obtained are in accordance with their analytical evaluations. Firstly, the types of parallelism exhibited by backpropagation neural systems were identified. Based on this, and as described in Chapter 4, the Abstract Decomposition module specified the alternative basic and multiple decomposition schemes of
backpropagation systems. Secondly, the targeted SIMD machine, the DAP was described in terms of both hardware and software. Thirdly, the analytical framework developed in this chapter was implemented to evaluate the speedup of the mapping schemes of backpropagation systems onto the DAP. This permits to rank the set of mapping schemes on the basis of the speedups they offer. The following mapping schemes were evaluated: the matrix-vector, the training, the matrix-matrix, and the distributed matrix-vector. The optimal scheme was identified on the basis of analytical criteria, these criteria involves both the neural system and the parallel machine (e.g. grain of mapping). Finally, the mapping schemes were implemented onto the DAP, and their execution was timed. This showed that the optimal mapping scheme experimentally retrieved is the same as the scheme selected by the analytical framework, allowing thus to validate the framework.
Chapter 7

ASN.1-based Specification for Neural Network Systems

This chapter introduces the Abstract Syntax Notation One (ASN.1) as the syntactic support for a specification for neural systems with explicit support for parallelism. It presents the ASN.1 concepts and demonstrates how these concepts are used for the description of neural systems.

7.1. Objectives

In Chapter 3, the need for a specification for neural systems which makes their parallelism explicit was demonstrated. Chapter 4 presented the Pygmalion nC specification for neural systems which incorporated parallel features. However, this specification had to be extended, in Chapter 5 in order to accommodate an automatic translation to MIMD machines. Following this, this chapter presents a formal specification for neural systems with explicit support for parallelism. The specification uses the Abstract Syntax Notation One (ASN.1) as a notational support. It is generic in that virtually any neural system can be expressed with it. The interest resides in the innovative use of ASN.1 for specifying neural systems; i.e. so far, ASN.1 has been dedicated to the specification of communication protocols. The objectives are thus to demonstrate the strength of ASN.1 as the notation for describing neural systems and their parallelism, and to show its usage.

This chapter is composed of three sections. The first section justifies the innovative use of the ASN.1 notation and introduces the concepts associated with. The second section shows how ASN.1 is used for specifying neural systems and their parallelism. An example of an ASN.1 structure of a neural model is also given. Finally, the third section summarises the integration and role of the ASN.1-based specification within the structure presented in Chapter 4.

7.2. The Abstract Syntax Notation

The Abstract Syntax Notation (ASN.1) is the internationally standardised way of describing high-level protocol information [Com88a]. It is a key ingredient of Open Systems Interconnections (OSI), and as such is widely used and recognised. A complete description of the Abstract Syntax Notation can be found in [Ste90a] provided by the CCITT (The International Telegraph and Telephone Consultative Committee). At this point, it should be noted that this section follows ASN.1 notation conventions; but might not indicate them[^14]. ASN.1 is a notation for describing structured information; typically

[^14]: In order not to confuse the reader with the rather strict syntactic conventions, these are
for specifying communication protocols. It allows arbitrarily complex structures to be built in a uniform way from simpler components [Ste90a]. Furthermore, it presents a complete divorce between information and its representation. With ASN.1 a designer describes the relevant information and its structure without being concerned at all with how it is represented.

The features of ASN.1 match the requirements for the neural specification: genericity, high level of abstraction, and explicit parallelism. Firstly, ASN.1 differentiates information with its representation; achieving thus the high level of abstraction needed by the specification. Secondly, ASN.1 is dedicated to the description of complex structured information. As shown in the previous chapter, neural systems are typically structured information. Thirdly, ASN.1 allows the construction of complex elements from simpler ones which is typically what a neural system is made of; i.e. a neural system consists of a structured set of basic elements (neurons and connections) which are used to construct complex elements; slab for example. Finally, the level of abstraction of ASN.1 permits the specification to encompass both data and processing information, at any level of the specification. This is necessary for a specification to be translated onto various parallel machines (both MIMD and SIMD).

In summary, ASN.1 allows a designer to consider its problem space at a high level of abstraction. Abstraction frees its user from engineering/implementation issues, and gives a global view of the element to be specified; neural systems in this case.

7.2.1. ASN.1 Types & Values

As mentioned earlier, ASN.1 is a notation for describing abstract structured information. ASN.1 defines a set of primitive data type and provides the facility to construct new elements with their own typing inherent in the structure. Abstract types and values form the basis of ASN.1. Types and values are interrelated notions; a type is a set of values, possibly infinite in size. A given ASN.1 type is an element of that type's set of values. Types (e.g. Boolean) represent potential information; values represent actual conveyed information (e.g. true, false). In the case of Boolean the set of values are restricted to two possibilities as opposed to, for example, the basic type PrintableString which has an infinite set of values (any sequence of printable characters).

There are two kinds of types: simple and composite. Simple types are the basic building blocks. For example BOOLEAN, PrintableString are both simple types. Composite type are defined in terms of other types; they are structured types made of simple types, or again structured types. For example, the ASN.1 composite type "SEQUENCE"
corresponds to an ordered collection of one or more types. In order not to overload the reader with unnecessary information, the full set of valid basic and composite types is given in [Ste90a]. A simple example which relates to neural systems is given below:

\[
NeuronStructInfo ::= \text{SEQUENCE} \{ \\
\quad \text{name-id[0]} \text{ PrintableString}, \\
\quad \text{elements[1]} \text{ INTEGER}, \\
\quad \text{data[2]} \text{ SET OF Parameter} \\
\quad \text{connections[3]} \text{ SET OF Connect} \\
\quad \text{processing[4]} \text{ NeuronStructEval} \\
\}\n\]

Figure 7.1 - ASN.1 Types: An Example -

This ASN.1 type NeuronStructInfo defines a type made of an ordered set of information items. NeuronStructInfo is composed of a name (character string), a number (e.g. number of its basic local processes), a set of Parameters (e.g. neuron's data), a set of Connects (i.e. another ASN.1 type defining the item Connect), and finally a set of NeuronStructEvals (ASN.1 type which specifies a neuron's computation). Parameter, Connect, NeuronStructEval refer to other ASN.1 types.

A module in ASN.1 is a named collection of types and values, it groups together a set of related definitions. However, the organisation of an ASN.1 specification into modules is entirely up to its designer; there are no restrictions such as "define before use" found in programming languages. Obviously, ASN.1 requires types used in a module and defined elsewhere to be declared as external, but this is the only requirement.

In summary, ASN.1 types govern a domain of possibilities; selecting a particular value corresponds to the choice of one particular possibility.

### 7.2.2. ASN.1 Macros

ASN.1 provides an additional mechanism; macro, which extends the abilities of defining new type and value notation\(^\text{16}\) [Con88a]. As defined in [Ste90a], macro is a mechanism within ASN.1 for its user to define new notation with which he/she can then construct and reference ASN.1 types or specify values of types. The definition of an ASN.1 macro specifies a new notation for referencing a type and also a new notation for specifying a value. Using macros, a user extends the notation for his/her own use, or for use by others. This is obviously one of the objectives of the specification for neural systems; i.e. its users are able to specify their own neural system. Macros are defined using the macro definition notation (MDN); each macro has a reference, as any ASN.1

---

\(^{16}\) Similarly to ASN.1 type and value, macro type and value are defined by giving their respective notation.
type, and grammars for type notation and value notation. These grammars are defined by
the designer. To an extent, macros can be regarded as adding another level of abstraction
to an ASN.1 specification. This is shown in the figure below, Figure 7.2 which presents
the macro mechanism.

![Diagram showing the integration of macros and ASN.1]

**Figure 7.2 - The Integration of Macros and ASN.1**

The definition of ASN.1 macros determine a structure space. After expansion, a
particular structure is determined; i.e. a particular ASN.1 structure is specified. Finally, by assigning values to the, now determined ASN.1 types, a specific assigned
structure is obtained.

As shown in Figure 7.2, ASN.1 macros allow the specification of a structure space. This
can be seen as defining a grammar which describes a set of valid ASN.1 structures. The
expansion of a macro is analogous to the description of a specific path in a set of
possibilities. It determines a specific ASN.1 structure; i.e. a structured set of ASN.1
types and possible values. Once this is done, values are assigned to each ASN.1 type,
delivering a fully instantiated structure.

Let us consider an example of an ASN.1 macro DEF_INTERACTION which defines the
interaction mechanisms by which a neural system interacts with its external environment,
i.e. application.
A macro user is able to use these new types (created by the macro) as any other types. The macro corresponds to the specification of concepts more complex than a data type, of which users can define instances. The type notation defines the template for assigning values to the ASN.1 types defined in the macro, with all the appropriate degree of freedom provided. In the example given in Figure 7.3, the macro permit the definition of three sorts of interactions which use the same ASN.1 types. The expansion of the macro determines the InteractionType which can either be a NEURON, SLAB or NETWORK. In abstract terms, there are no differences between a neural application that interacts at a slab level or at a neuron level.

In summary, ASN.1 provides two levels of instantiation: the expansion of macros that sets a particular structure of ASN.1 types, and the ASN.1 value assignment that assigns each value of each type. These concepts are now applied to the specification of neural systems.

### 7.3. Neural Systems in ASN.1

This section shows how ASN.1 and macros are used as the notation for the specification of neural systems. Firstly, it describes neural systems as structured information; this is based on the framework developed in chapter 2 and 3 which incorporates parallelism. Secondly, it specifies a simple example of a neural model specified in ASN.1. This example includes both ASN.1 types and their assignment to specific values. Lastly, it shows how this is generalised in order to construct the ASN.1 types and values of any neural system.
7.3.1. Neural Systems as Structured Information

Neural systems are defined as the combination of a neural model, application and system processing. The types of parallelism exhibited by neural systems have been classified into system-related and model-related. From this, any neural system and its types of parallelism can be specified as a hierarchical structure primarily formed by the composition of an application, system processing, and a neural model. Each of these elements may exhibit a type of parallelism, and is composed of sub elements, sub elements which may in turn exhibit a type of parallelism. Table 7.1 summarises the structure of a neural system, this is based on the framework for neural systems developed in Chapter 2 and the classification of their parallelism done in Chapter 3.

<table>
<thead>
<tr>
<th>Neural System</th>
<th>Types of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Data to &amp; from the model</td>
</tr>
<tr>
<td>System Processing</td>
<td>Control of the Network Dynamic Processing</td>
</tr>
<tr>
<td>Neural Model</td>
<td></td>
</tr>
<tr>
<td>Global Elements</td>
<td>Network Dynamic Processing</td>
</tr>
<tr>
<td></td>
<td>Network Interconnectivity</td>
</tr>
<tr>
<td></td>
<td>Organisation of Network of Neurons</td>
</tr>
<tr>
<td>Local Elements</td>
<td>Neuron's Data</td>
</tr>
<tr>
<td></td>
<td>Neuron's Computations</td>
</tr>
<tr>
<td></td>
<td>Connection's Data</td>
</tr>
<tr>
<td></td>
<td>Connection's Computations</td>
</tr>
</tbody>
</table>

|                  | Inter-slab                      |
|                  | Intra-slab (Inter-connection)   |
|                  | Intra-slab (Inter-neuron)       |
|                  | Intra-neuron                    |

Table 7.1 - Neural Systems as Structured Information -

The top elements of the structure consists of a neural application, the system processing, and a neural model. Each of these top element is then further divided. The system processing defines the transforms the model applies in accordance with the application. Both the application and the processing can exhibit parallelism, as described in Chapter 3. The neural model divides into global and local elements. The global elements define the organisation of the network of neurons, the interconnectivity pattern, and the dynamic behaviour of the network. The local elements specify the neurons and connections. They determine their type and local computations. Each of these elements can exhibit parallelism. For example the dynamics of the network if parallel defines the inter-slab type of parallelism (as defined in Chapter 3).

A neural model is expressed hierarchically, the hierarchy determines the model's organisation; e.g. existence of different slabs performing different operations. Each level, in the hierarchy, expresses data and processing information, at its own level and is dependent on the levels it is composed of. For example, a model can be formed by two slabs of neurons and one slab of connections, and the learning is performed by applying transforms $x_1, x_2$ respectively on the slabs of neurons, followed by a transform $\gamma$ on the connections. This structure makes explicit the parallelism exhibited by each level by indicating the type (parallel or sequential) of each computation. An example is now considered.
7.3.2. A Neural Model ASN.1 Types & Values

The following describes a two layers neural model fully interconnected which performs a standard forward phase.

Figure 7.4 - Structured Representation of a Two-Layers Neural Model & its Computations - The left-hand side of the figure describes the neural network model. The figure in the middle displays the interconnectivity pattern and the organisation of the network of neurons as a structure. Finally, the right-hand side presents the dynamic processing operated by the model during an hypothetic forward phase, it is represented as a tree structure.

The hierarchical representation divides the function and data of neural network systems. The example shown in Figure 7.4 presents a two-layers fully connected neural model and its computation. The left-hand side of the figure displays the neural model in a standard manner, i.e. neurons are depicted as circles and connections as arrows.

The middle picture of Figure 7.4 shows the organisation of neurons and connections. The model consists of two slabs of neurons, denoted S1 and S2, each of which respectively contains three neurons (n1, n2, n3) and two neurons (n4, n5). Each neuron from S1 is connected to each neuron from S2.

The right-hand side of the figure considers the forward computation phase performed by the neural model. The global computation, i.e. computation operated by the model divides into two slab computations, S1_forw and S2_forw. The computations performed by both slabs exhibit parallelism, i.e. the collection of neurons’ computations they specify can occur in parallel. The neurons n1, n2, n3 are set to values x in parallel, and the neurons n4, n5 perform the standard computation of their net input: $\sum_j w_{4j} x_j$ for n4, and $\sum_j w_{5j} x_j$ for n5 also in parallel. This is shown at the bottom right-hand side of Figure 7.4.
The ASN.1 specification of this neural model is described below, it uses the NeuronStructInfo type and the INTERACTION macro described earlier. The ASN.1 specification embodies both the structure of the model, i.e. the organisation of neurons and connections, and the computations performed by the model, a forward phase in this case. A neural system so described can thus be decomposed either by analysing its computations, i.e. leading to a processing-based decomposition, or by analysing its structure, i.e. leading to a data-based decomposition.
-- Definition of the ASN.1 types for specifying the two-layers neural model
-- and its computations.
-- These types are obtained via the expansion of ASN.1 macros

NNModel ::= SEQUENCE {
  data Data
  infostruct SEQUENCE OF Slab
  compstruct Eval }

Slab ::= SEQUENCE {
  slabdata SlabData
  slabinfostruct SEQUENCE OF NeuronStructInfo
  slabcompstruct SlabStructEval
  interaction INTERACTION }

Data ::= ANY

Eval ::= SEQUENCE {
  processingtype Enum {seq , par }
  compstruct SEQUENCE OF SlabStructEval }

SlabStructEval ::= SEQUENCE {
  processingtype Enum {seq , par }
  compstruct SEQUENCE OF NeuronStructEval }

NeuronStructEval ::= SEQUENCE OF {
  indata SEQUENCE OF PrintableString
  outdata SEQUENCE OF PrintableString }

-- Assignment of the ASN.1 types, i.e. ASN.1 values representing the specific
-- neural model (made of two slabs, each containing respectively three and two
-- neurons) and its computations, i.e. forward computation.
-- In ASN.1 values are assigned as follows: ValueName ASN.1Type = Value

M1 NNModel = { {}, {S1, S2} , p }
M1 Eval = { seq , {S1.forw, S2.forw} }
S1 Slab = { {}, {n1, n2, n3} , S1.forw, interaction }
S1.forw SlabStructEval = {par, {n1.clamp, n2.clamp, n3.clamp} }
S2 Slab = { {}, {n4, n5} , S2.forw, {} }
S2.forw SlabStructEval = {par, {n4.forw, n5.forw} }

-- The assignment of the neurons is made in accordance with the NeuronStructInfo
-- ASN.1 type defined earlier.

n1 NeuronStructInfo = { "N1", 1, {x Float}, {}, n1.clamp }
n1.clamp NeuronStructEval = {x Float} -- Clamp x to external value

-- similar assignment for n2 and n3

n4 Neuron = { "N4" , 1, {x Float}, W, n4.forw }
W = { w1 Float, w2 Float, w3 Float }
W n4.forw NeuronStructEval = { ("W"."x") {"res"} }
  -- res := dot product W by x from S1

-- similar assignment for n5

In general, the high level transform operated by a system is expressed as a
parallel/sequential set of lower level transforms along with the data it applies to, and so
on until it reaches the neurons and/or connections level. The action of representing
neural systems as a set of structured information is equivalent to a kind of preprocessing
for parallelisation; it makes explicit the parallelism both in data and processing. We now
discuss how this structural view of neural systems is obtained using ASN.1 formalism.
7.3.3. Usage of ASN.1 for Specifying Neural Systems

The ASN.1 acts as the notational support for the specification of neural systems. In other words, ASN.1 is used as a formalism for specification purpose, it is not a true programming language. A neural system is represented by an instantiated ASN.1 structure. The following figure, Figure 7.5, presents the stages to follow in order to construct this ASN.1 structure representing a neural system.

![Flowchart showing stages to construct an ASN.1 neural system]

**Figure 7.5 - The Construction of a Neural System using ASN.1** -
The set of neural models is defined via macros; this is analogous to the specification of a grammar. These macros are then expanded to obtain a specific neural model; this is analogous to the determination of a specific path within the grammar. The ASN.1 types obtained are finally instantiated along with the ASN.1 description of application (cf INTERACTION) and system processing.

As shown in Figure 7.5, the three stages to construct an ASN.1 neural system are:

- Define a Set of Neural Models via ASN.1 Macros
- Determine a Neural Model via the Expansion of Macros
- Determine a Neural System via the Assignment of the ASN.1

**Define a Set of Neural Models via ASN.1 Macros**

The space of neural models is defined via ASN.1 macros. The macros, i.e. grammar allows the specification of any valid neural model. Models can be structured or unstructured, homogeneous or heterogeneous. They can be composed of a set of neurons and connections, or a set of slabs, each of which made of neurons and connections. Macros define each basic type of processing, for each level (if any) in the model’s organisation. For example, a macro could be written to specify a standard activation computation. At this stage of abstraction, the processing types mainly differentiate sequential processing versus parallel processing. It is only when setting the neural system that the processing is refined and its parameters determined. A prototype of ASN.1 macros for defining the space of neural models is given in Appendix 7.1.
Determine a Neural Model via the Expansion of Macros

The specification of a neural model is the result of the expansion (instantiation) of the ASN.1 macros. A neural model is created by setting its organisation and its local and processing. The neural model and its forward phase presented in Figure 7.4 represents the result of an expansion. At this stage, the processing of the model, e.g. forward phase, is specified and the information required by each level is determined. The result of the expansion of ASN.1 macros have been presented in the example above.

Determine a Neural System via the Assignment of the ASN.1

The assignment of the ASN.1 structure defines a specific neural system. Firstly, the structure of ASN.1 types obtained (after the expansion of the ASN.1 macros) is assigned following the value notation defined in the macro. Secondly, the ASN.1 types representing the neural application and system processing are assigned, this gives a complete ASN.1 structure of a specific neural system.

4.4. Summary

This chapter has introduced the usage of ASN.1 for the specification of neural systems. It has described how the space of neural models can be defined using the ASN.1 macro mechanism. Then, from the expansion of macros, it has explained how a specific neural model is obtained. Finally, it has shown how the assignment of the ASN.1 types builds a specific neural system. By taking ASN.1 as the formalism for describing neural systems no assumptions are made regarding the machine the neural system will execute on. The ASN.1 is not a programming language and is not executable, it always requires translation/transformation into an appropriate format for execution (either on sequential or on parallel machines). Furthermore, neural systems expressed as ASN.1 structure make explicit both their data structure, i.e. hierarchy of neurons and connections and slabs, and the structure of their computations, as shown in the ASN.1 example given in the previous section. This makes possible decompositions of neural systems based on either their computations or data structure.
Chapter 8
Assessment

This chapter assesses the work accomplished in this thesis. The assessment covers five areas: (i) the analysis of neural systems and parallel machines, (ii) the System for decomposing and mapping neural systems onto general-purpose parallel machines, (iii) the MIMD mapping prototype, (iv) the experiments carried out on an SIMD machine, and (v) finally the ASN.1-based specification for neural systems.

8.1. Neural Systems & Parallel Machines

This thesis focused on the problem of decomposing and mapping neural systems onto general-purpose parallel machines. Neural systems and parallel machines were analysed with two objectives in mind:

- To show that programming and executing neural systems require mechanisms with explicit support for parallelism,
- To identify the properties of a mechanism to support the optimal mapping of neural systems onto general-purpose parallel machines.

Both objectives are met: the analysis of neural systems declared the motivations for providing mechanisms to express and exploit the parallelism they exhibit, and the analysis of parallelism in neural systems and machines identified the properties of such a mechanism.

As stated in Chapter 1, the study and practice of neural systems involves a large amount of experiments, and is thus dependent on the availability of environments for neural systems. Existing environments and/or implementations of neural systems that express and exploit parallelism were studied. This revealed weaknesses in current systems for describing and exploiting the parallel nature of neural systems. The spectrum of neural systems and general-purpose parallel machines was surveyed, and their main characteristics were identified.

Following this, parallelism in neural systems and machines was investigated. The key issue is to map the parallelism of neural systems onto the machine’s parallelism. The parallelism exhibited by neural systems was formalised into types: neural system related types and neural model related types. In doing so, it demonstrated that the framework presented in Chapter 1 was able to specify neural systems as well as their types of parallelism. Parallel machines and the different types of parallelism they support (determined by both their architecture and their software support) were discussed. Based on this, the decision was to opt for a specification for neural systems which provides
abstraction for parallelism and is machine independent, and also to provide mechanisms that efficiently transform it for execution onto a targeted machine. These mechanisms were required to adapt to both neural systems and machines.

8.2. System for Decomposing & Mapping Neural Systems

The objectives of the System for decomposing and mapping neural systems were identified in order of importance as follows:

- generic - able to map virtually any neural system onto any parallel machine,
- transparent - the mapping schemes of neural systems are automatically produced.
- optimal - the mapping produced is optimal for the specific neural system and machine.

The System developed in Chapter 4 and 6 achieves these objectives. Firstly, it adapts to various neural systems and machines, showing thus genericity. Secondly, alternative mapping schemes of a neural system are analytically evaluated in order to implement the optimal scheme. Thirdly, the mapping schemes are potentially automatically produced; this is demonstrated by the implementation of a prototype software.

To achieve genericity with respect to neural systems and parallel machines, the System divides the mapping task into a machine independent task: the Abstract Decomposition (AD) which deals with neural systems and their alternative decomposition schemes, and a machine dependent task: the Machine Dependent Decomposition (MDD) which integrates the decomposition schemes of a neural system with a targeted machine in order to generate the specification of the optimal mapping scheme.

The AD formalises the various ways for decomposing neural systems, which is currently done in an ad-hoc way. Furthermore, the AD is capable of dealing with various neural systems, and extracts all their available parallelism. Unfortunately, given the constraints of machines, it is not always possible to exploit all such parallelism. In this research, the AD specifies the decomposition schemes of neural systems in a machine-independent manner, and thus provides all possible alternative decomposition schemes of neural systems.

The MDD determines the alternative decomposition schemes of neural systems onto the targeted parallel machine, i.e. the mapping schemes. It developed an analytical framework which is able to analytically evaluates the speedups of alternative mapping schemes, and for virtually any neural system and parallel machine. The automatic generation of mapping schemes for neural systems is validated by the MIMD prototype which design corresponds to the System for decomposing and mapping. It is evaluated in the next section.
8.3. MIMD Prototype

This section evaluates the MIMD mapping software prototype described in Chapter 5. Firstly, it shows that the prototype meets the objectives of both this thesis and the Pygmalion project for which it was originally built. Secondly, it compares the mapping software with existing software described in the literature.

8.3.1. Achievements

With respect to the Pygmalion project, the prototype's objectives are met. The developed mapping software enables a Pygmalion user to automatically generate a neural system that executes on both sequential machines and Transputer-based machines. Furthermore, the functionalities available to the user are equivalent wherever the simulation occurs; i.e. on a host workstation (sequential) or on a Transputer-based machine (parallel) [Aze90a].

With respect to this dissertation, the prototype's objectives are met. The construction of the prototype validated:

- the System for decomposing and mapping neural systems,
- the feasibility of an automatic mapping of neural systems into their parallel equivalent.

The design of the prototype corresponds to the System for decomposing and mapping developed in Chapter 4. The mapping software implemented consists of an Abstract Decomposition and a Machine Dependent Decomposition. The AD analyses neural systems transforms their specification into a set of communicating tasks, it implements a processing-based decomposition. The MDD module selects the appropriate mapping scheme according to a set of rules. The nC specification of a neural system is finally transformed into its parallel equivalent. The mapping software applies to virtually any neural system, and is able from the nC specification of a neural system to produce the equivalent system executable onto the Parsys SN1000 [Aze91a]. Within the Pygmalion project, the implementation has been tested for the Competitive Learning, Backpropagation, and Kohonen Self-organising Map neural models. Lastly, it should be noted that the relative bad performances obtained in the experiments carried out are mainly due to the application used, and the insufficient study of the deficiencies of the 3L Parallel C + Tiny software used during implementation, as justified in section 5.4 of Chapter 5.
8.3.2. Comparison with Existing Mapping Systems

Several researchers have implemented neural systems on MIMD distributed memory machines, and more particularly onto Transputer-based machines. The work done in this area can be divided into four groups:

(1) - implementation of an application of a specific neural system on a specific Transputer-based machine [Zha90b],
(2) - implementations of a neural model on Transputer networks, [Ult89a], and [Pet91a],
(3) - implementations of a class of neural models on Transputer networks [Bey89a], and [Aik90a],
(4) - implementations of classes of neural systems on Transputer networks [Etn90a], and [Mik90a], and [Koi89a].

The group (1) describes highly dedicated neural systems mappings; for example the implementation of the Traveling Salesman problem on a Meiko Parallel Computer [Zha90b]. This type of implementations is non-adaptive; they are targeted to one machine and one neural system and thus one application. It uses particularities of both the application and the neural model. In the previous example, each processor is assigned to a single number of cities. This is a direct consequence of the application, and it is not related to the simulated annealing process the neural system performs. The prototype software developed in Chapter 5 is not restricted to particular neural systems.

The group (2) describes mappings which can adapt to different configurations of neural models. For example, Petrowski presents a parallelisation of the backpropagation neural model for Transputer networks. In this group, the methods for parallelisation adapt to various configurations of a specific neural model. In [Pet91a], the parallelisation devised is such that the computation efficiency weakly depends upon the configuration of the network. However, this parallelisation only exploits a particular type of parallelism of the neural model concerned; i.e. the training parallelism. In our case, the prototype software can exploit different types of parallelism according to the targeted Transputer network and to the neural model and its configuration.

The work done in group (3) is an improved version of group (2) in that the parallelisation adapts to a class of neural models. For example, Beynon in [Bey89a] describes two kinds of neuron-to-Transputer allocations which apply respectively to fully connected or sparsely connected multi-layer perceptron type neural models. However, none of these groups; (1), (2), and (3) provide automated mechanisms for either parallelisation as such, or transformation from sequential to parallel specification. In this respect, the prototype software is superior; i.e. it provides an automatic transformation from a sequential specification into its parallel equivalent [Aze91a].
The group (4) describes mapping software dedicated to Transputer-based networks. Such mapping software is the closest to the prototype; i.e. they are able to provide parallel specifications of virtually any neural system for Transputer networks [Koi89a], and [Ern90a]. However, the design of such software embeds machines' features. For example, the Carelia simulator [Koi89a] provides a specification for neural systems that directly maps into Occam and as such embeds knowledge about Transputers. Occam and Transputers are intrinsically related. In this respect, these mapping softwares are tied to a particular type of parallel machine. Instead of this, the prototype's design is not targeted to a parallel machine. The design divides the mapping task into a machine independent module and a machine dependent module. The prototype via its Abstract Decomposition module embeds no knowledge about the targeted parallel machine. It devises the different parallelisation methods; i.e. decomposition schemes independently of the machine's characteristics which are taken into account at a later stage by the Machine Dependent Decomposition module. Based on this, the prototype is generic in that its design virtually applies to any type of general-purpose parallel machines.

8.4. Analytical Framework on SIMD

This section evaluates the SIMD experiments illustrating the analytical framework described in Chapter 6. Firstly, it assesses the experiments in the context of this thesis. Secondly, it presents various approaches for mapping neural systems onto SIMD type machines described in the literature. It compares them and the results obtained to the one reported in this chapter, and concludes on its benefit.

8.4.1. Achievements

Chapter 6 described the alternative mapping schemes of backpropagation neural systems onto a typical SIMD machine, the DAP machine. It firstly specified the decomposition schemes of backpropagation systems. Secondly, taking into account the characteristics of the targeted parallel machine, the MDD module analytically retrieved the most suitable mapping scheme. This was achieved by evaluating the speedup of the alternative mapping schemes, and by selecting the mapping scheme offering the best one. The framework was validated by showing that the analytical ranking of mapping schemes is equivalent to their experimental ranking. The analytical and experimental stages were as follows:

- Analytical
  
  Use the framework to analytically specify the expected speedups of various mapping schemes of backpropagation neural systems onto the DAP,
Retrieve the mapping scheme with the maximal speedup.

- Experimental
  
  Implement the various mapping schemes onto the DAP and time their executions,
  
  Rank the schemes accordingly to identify the optimal scheme.

The analytical and experimental rankings of the mapping schemes agree with each other, validating thus the analytical framework.

8.4.2. Comparison with Existing Implementations

The work described in the literature can be divided into three groups:

(1) - mapping specific neural systems onto a specific SIMD machine [Ros90a],

(2) - mapping specific neural systems onto various SIMD type machines [BroSSa], and [TomSSa],

(3) - developing SIMD architecture for implementing neural network [WitS9a], and [Bli91a].

The group (1) describes implementations (mappings) of neural systems which exploits particular hardware features of a targeted parallel machine. A good example is the implementation of backpropagation neural systems onto the Connection Machine described in [Zha90a]. This implementation is analogous to the matrix-matrix scheme implemented, but where the authors exploit the features of the machine hardware. The Connection Machine has the particularity of possessing a shared memory for 32 processors [Hil85a]. They use this facility to store the connection weights of connecting layers so that all the processors in this group update their weight in parallel, this is analogous to a partial matrix-vector mapping. Another example is described in Rosenberg et all [Ros90a] which presents a mapping of backpropagation system in which each neuron is allocated to one processor and each weight to two processors (the weight is then duplicated as weight-in and weight-out). The implementation is further improved by overlapping the weight-in and weight-out and by pipelining the layers. That is, given a set of $n$ input vectors and $m$ layers $l_i$ they propagate the $i^{th}$ input vector across the first layer while propagating the previous input vector $v_i - 1$ across the second layer, $v_i - 2$ across the third layer and so on until the last layer. In our case, the particularities of the DAP machine were not used. The analytical framework evaluates expected speedups on the basis of standard features of the machine; e.g. grain and communication time. In this respect, the framework can virtually apply to any parallel machine [Aze92b] to forecast speedups of different mapping schemes.

---

17 This requires some accurate bookkeeping so that no two copies of the network tries to write the same weight at the same time.
The work done by the group (2) improves the work described in the first group by presenting the mapping of specific neural systems but onto a range of SIMD machines. For example, Zhang and all in [Zha90a] describes a model for mapping backpropagation neural systems onto massively parallel architectures which have 2D or higher degree connections among their processors. Their model uses the 2D nearest neighbour communication facility offered by this type of machine, and implements a combination of net replication (parallelise the training) and connection-layer (parallel layer weight update) decompositions. Others have worked with the assumption that using an SIMD machine implies an equivalence between neuron and processor. In these cases, e.g. [Tom88a], the problem resides in the representation of the connections between neurons. Hardware limits the number of physical communication channels between processors. Specialised software routing methods have been developed, however these methods even though quite efficient for big (in terms of neurons) and sparsely connected neural systems, are acknowledged as inefficient for highly connected neural systems. The work presented does not make this assumption of equivalence between neuron and processor, i.e. the selected mapping scheme might take this approach but this depends on both the neural systems and machine characteristics. Furthermore, the framework is not dedicated to the backpropagation systems; i.e. the framework applies to other neural systems [Aze92b]. This is also true for the targeted machine, i.e. the analytical expressions adapt to the machine.

Lastly, the work done group (3) can be divided into two sub-groups. The first one concerns the development of neural software on experimental SIMD machines. The work done by Witbrock and al is a typical example of the first sub-group [Wit89a]. They developed a neural network simulator for the GF11 SIMD machine, an experimental parallel computer [Bee86a] developed at IBM’s Watson Research Centre. Their underlying aim was to speed up the execution of backpropagation systems for speaker-independent continuous speech recognition tasks [Fra89a]. They conclude that, although the GF11 was conceived with Quantum Chromodynamics applications in mind, the architecture is quite well suited for, at least, the implementation of backpropagation systems. The second group represents people that specifically designed parallel machines for simulating large neural systems. For example, Elias and al [Eli91a] present an experimental multiprocessor machine with SIMD-like architecture and enough memory (both local and distributed) to store the vast amount of data and parameters used in large networks. These machines are usually constructed so as to speedup the standard computations occurring in neural systems; e.g. connection weights computations, sum of products and non-linear functions. These specifically designed parallel machines will offer better speedups for neural computations than the approach presented. However, our
aim was to balance genericity against speedups. The framework for mapping neural systems provides a generic approach for selecting the best mapping scheme associated with a neural system and a parallel machine. Thus, with respect to genericity the approach implemented here is more powerful.

8.5. Specification for Neural Systems

This section assesses the specifications for neural systems introduced in this thesis, namely nC and ASN.1-based. However, it concentrates on the ASN.1-based specification. The reasons for this are as follows. Firstly, the implementation of the mapping software described in Chapter 5 uses the nC specification, and as such demonstrates its validity. Secondly, the nC specification was developed by the Pygmalion team [Aze90b], while this is not the case for the ASN.1-based specification. This section is thus divided into two parts: firstly, the evaluation of the ASN.1-based specification versus other existing neural systems languages, and secondly the comparison between the ASN.1-based and nC.

8.5.1. ASN.1-based Specification

In addition to the System for decomposing and mapping neural systems, i.e., AD and MDD modules, a specification for neural systems was introduced. This specification uses the Abstract Syntax Notation One (ASN.1) as the syntactic support, and provides explicit parallelism. Many specifications or languages dedicated to neural systems are based on traditional sequential languages, for example Neural Pascal [Gum90a]. Furthermore, existing languages are bound to simulation environment, for example the AXON language is dedicated to the HNC environment [Hec90a], or the C-based specification developed for the UCLA-SFINX environment [Pai87a]. We, however believe that languages should separate concerns between the specification and the implementation. This is obviously critical if the specification is to be implemented onto a range of machines, which is of prime concern in this thesis. This thesis introduced the ASN.1 notation for specifying neural systems, and demonstrated its usage. This notation is not a programming language, it is thus difficult to compare it with existing languages. It is however similar to existing one in that it represents neural systems as structured information. A neural system is composed of a neural model, application and system processing, and is then further divided into smaller elements until neurons and connections (the basic elements of neural systems) are defined. This way of representing neural systems is used in nC [Aze90b], AXON [HNC89a], and in the MetaNet environment [Mur91a]. These languages have demonstrated their validity within the neural computing community; for example nC and its associated environment Pygmalion has been released to the public. This enforces the validity of the ASN.1-based
specification. The principal feature of the ASN-1-based specification is that it supports explicit parallelism in a machine-independent manner. This is not the case of AXON, for example. MIND developed by Koikkalenen uses an extension of the Communicating Sequential Processes (CSP) formalism, adding neural network related features [Koi91a]. In this respect, MIND stands at a low architectural level. This thesis, with the introduction of ASN.1 as the formalism for neural systems, took a different approach by putting the emphasis on the abstraction. This permitted us to make no assumptions about the targeted neural system and machine. We believe this is paramount when one develops mechanisms enabling the execution of any neural system onto any machine, and more particularly parallel machine.

8.5.2. ASN.1 & nC

The purpose of this section is to establish the similarities between nC and ASN.1, and to show how the ASN.1-based specification improves the expression and potential exploitation of parallelism of neural network systems.

Historically, the nC specification format was developed before the ASN.1-based specification. The nC specification was built by the UCL team of the Pygmalion project as a target language to accommodate the requirements of neural network programming environments [Aze90a]. On the other hand, the ASN.1-based specification corresponds to the initial step of the System for decomposing and mapping neural systems onto general-purpose parallel machines. Because of the differences in goals between the two, they tackle the problem of specifying neural systems from different angles. Firstly, nC was built to be executed on sequential machines as opposed to the ASN.1-based specification which requires transformation for implementation on either sequential or parallel machines. Secondly, the ASN.1-based specification puts the emphasis on the expression of parallelism as opposed to the nC specification which had also to take into account the Pygmalion environment constraints. For example, the nC rules needed to be monitored and executed from the Pygmalion the Graphic Monitor. It should be noted that the ASN.1-based specification is not to be translated into an nC equivalent, it is the constructed neural system that are similar. That is, the memory image of a neural system when constructed via the extented nC is equivalent to its transformed ASN-1 representation.

Both the nC and ASN.1-based specifications view neural systems as hierarchical set of information, even though the level of abstraction of each is different (cf Chapter 4 & Chapter 7). Indeed, the ASN.1-based specification makes no assumptions on how each level is to be represented when implemented, while nC, because it is a subset of C, needs to specify a valid C data type. Each ASN.1 type can be implemented into virtually any C type, or types of other programming languages. Another similarity is that each
specification indicates if the processing occurring in a neural can be performed in parallel or in sequence. This is described via the *ProcessingType* in ASN.1 notation, and via the *pexec, sexec* types in nC. Furthermore, each processing specifies the components it acts on. In ASN.1 notation, this was described as *SlabData* for a slab, or *Data* for a neuron. In nC, it was described via the *rules parameters* type of a rule. However, this rule parameters type had to be extended to accommodate an automatic translation onto MIMD machine. Indeed, it was not possible to extract the data changed by a rule as opposed to accessed by a rule, which is paramount for translation to MIMD-based machines. The ASN.1-based specification was proposed with the specific goal to express parallelism in a way suitable for allowing translation to any type of parallel machines, and as such to provide all required information. This is intrinsic to the ASN.1 notation in that it is not a programming language, e.g. it does not refer to pointers as nC (based on the C language) does. ASN.1 data types are abstract data types, and no assumptions are made concerning their representation. Thus, processing (or rules in the context of nC) is an abstract data type as any other, while in nC it is already tighten to a "C representation". This implies, for example, the nC pointers to be unwind for translation into vector processors parallel machine.
Chapter 9

Conclusion

This last chapter summarises the thesis, and the methodology followed. It also outlines prospects for future work.

9.1. Thesis Summary

This thesis developed a Mapping System for decomposing and mapping neural network systems onto general-purpose parallel machines. The challenges addressed and solved by this thesis are:

The need for mechanisms to express and exploit the parallelism inherent in neural systems was demonstrated.

Chapter 2 investigated neural systems and developed a generic framework for describing them. It showed that the nature of programming and executing neural systems is oriented towards parallelism. It then surveyed parallel machines, and more particularly their architectures. Finally, it revealed weaknesses in current environments in supporting the expression and exploitation of the parallelism exhibited by neural systems.

The requirements for a Mapping System for decomposing and mapping neural systems were identified.

Chapter 3 identified the requirements for a System for mapping neural systems onto general-purpose parallel machines by analysing the parallelism both in neural systems and machines. The parallelism exhibited by neural systems was classified into types, and integrated within the descriptive framework developed earlier. Parallel machines; their parallelism, their languages and the approaches to deal with their exploitation were analysed. Based on these analysis, the approach taken in this thesis was justified.

A Mapping System for decomposing and mapping neural systems onto general-purpose parallel machines was proposed.

Chapter 4 developed a System for decomposing and mapping neural systems onto general-purpose parallel machines. The System divides into an Abstract Decomposition module and a Machine Dependent Decomposition. The nC specification for neural systems with parallel features was described. The AD module specified the set of alternative decomposition schemes of neural network systems relative to the types of
parallelism they exploit. The MDD module analysed the set of alternative decomposition schemes in conjunction with the features of parallel machines, and selects the most suitable one for automatic translation.

A mapping software prototype onto a typical MIMD type machine was implemented. Chapter 5 described the implementation of a system that automatically maps static neural systems onto a representative example of MIMD machines, a Transputer-based machine. This prototype automatically decomposes neural systems, and generates the parallel specification of an optimal mapping scheme for the 48-processor Parsys SN1000 Transputer-based machine.

An analytical selection of the optimal mapping scheme for a neural system onto a typical SIMD machine was developed and validated. Chapter 6 developed an additional tool for the MDD module of the System. This tool, an analytical framework, was able to evaluate the speedup of alternative mapping schemes of neural systems onto parallel machines. It permitted the selection of an optimal mapping scheme for the implementation of a neural system on a parallel machine. The validity of the analytical framework was also demonstrated. This chapter gave the implementation of various schemes for mapping backpropagation neural systems onto an SIMD machine, the Distributed Array of Processors. It verified that the analytical ordering of neural systems mapping schemes is equivalent to the one obtained experimentally (by implementing and timing the mapping schemes).

An ASN.1-based specification for neural systems was introduced. Chapter 7 proposed a specification for neural systems with explicit support for parallelism. This specification introduced the Abstract Syntax Notation One (ASN.1) as the notational support. The innovative use of ASN.1, previously dedicated to the specification of communication protocols, was justified, and its usage for describing neural systems and their parallelism was described.

This thesis developed a solution for decomposing and mapping neural systems onto general-purpose parallel machines. It described a System for the decomposition and mapping of neural systems. The System is general in two distincts ways:
- It is applicable to virtually any neural system and general-purpose parallel machine. This generality is embodied in the lack of restrictive assumptions about the neural systems and parallel machines.
- It provides the stages to be followed in the design and implementation of a software for mapping neural systems onto a parallel machine.

Mapping methodology and tools, such as the one developed in this thesis, invariably perform less well than a specific mapping tool. A hand-coded implementation of a backpropagation neural system onto Transputer-based machine is expected to be faster than an automatic implementation, because it can take advantage of the specificities of the machine. This research established a compromise between speedup and genericity. With respect to genericity the System for decomposing and mapping adapts to various neural systems and parallel machines. With respect to speedup the System, via its MDD module, is able to select an optimal scheme according to the neural system and its parallelism, and according to the parallel machine. The selected scheme is optimal with respect to the set of alternative mapping schemes the System identified.

This dissertation has therefore provided novel general-purpose contributions to the field of neural computing and general-purpose parallel machines:
- a System for decomposing and mapping neural systems onto general-purpose parallel machines,
- an analytical framework for evaluating speedups of alternative mapping schemes of neural systems onto parallel machines,
- a notational construct for specifying neural systems with explicit support for parallelism.

9.2. Future Work

Both the research and the experiments carried out in this thesis can lead to further work. Three prospects for future work are outlined, these are:
- Upgrade of the MIMD prototype,
- Extension of the usage of the Abstract Syntax Notation One (ASN.1),
- Development of a unified formalism for parallel machines.

As indicated in Chapter 5, the prototype mapping software for MIMD machines automatically maps static neural systems. At this stage of implementation the prototype does not support the mapping of constructive neural network systems [Fal87a], i.e. systems whose configuration modifies during simulations. This can be solved by integrating within the prototype a dynamic load balancing tool to deal with constructive systems [Lin87a]. In this case, the Machine Dependent module incorporates a dynamic-load balancing tool which, combined with the current MDD, will be able to provide a mapping scheme specification which dynamically adds or deletes the appropriate neuron, slab or network tasks. The implemented mapping software is adaptive in that new components can be easily added.
This thesis developed a System for decomposing and mapping neural systems, and introduced the use of the Abstract Syntax Notation One (ASN.1) for specifying neural systems and their parallelism. The ASN.1 notation could also be used to describe the various decomposition schemes of neural systems. This is because ASN.1 is by definition dedicated to the specification of complex data information system, and thus can apply to various applications, neural system decomposition schemes in our case. The ASN.1 formalism would thus be used throughout the machine-independent part of the System (the Abstract Decomposition). In doing so, the implementation of neural system mapping schemes would be transparent and would only require the transformation of the ASN.1 description of the decomposition schemes into the parallel machine's format. Furthermore, as indicated in Chapters 7 and 8, ASN.1 is a formalism which needs, in any case, to be translated into an appropriate programming language in order to be executable.

Concerning the machine-dependent part of the System, it would be challenging to attempt to provide a formalism to describe parallel machines. As indicated in Chapter 3, there is as yet no unified framework for parallel machines. A possible starting point for the development of such a formal and/or analytical specification could be the bridging model between software and hardware for parallel computation developed by Valiant in [Val90a]. Valiant's work advocates (as we do) the need for a model analogous to the Von Neumann model for parallel computation, if it is to become widely used. The model introduced, called the Bulk Synchronous Parallel (BSP) model, has proved to be appropriate by quantifying its efficiency both in implementing high-level language features and algorithms, as well as in being implemented in hardware.
References

Aar89a.

Aar87a.

Agh86a.

Aik90a.

Akl89a.

Ama77a.

AMT90a.

And89a.

Ara86a.

Ath88a.

Aze90a.

Aze92b.

Aze91a.

Aze90c.

Aze92a.


Com88a.

Con88a.

Dun90a.

Eli91a.

Ern90a.

Fal87a.

Fly66a.

For87a.

Fra89a.

Gal89a.

Gib89a.

Gro88a.

Gum90a.

Gup90a.
Han78a.

Hec90a.

Hil85a.

Hin87a.

HNC89a.

Hoa78a.

Hoc87a.

Hoc84a.

Hol92a.

Hop82a.

Hwa84a.

INM84a.

J.B90a.

Joh90a.

Kal90a.

Kno89a.

Koh87b.

Koh87a.
Koi91a.

Koi89a.

Kun87a.

Lin87a.

Lip87a.

May88a.

Mig90a.

Mik90a.

Mur91a.

Nes88a.

Obe90a.

Pai87a.

Par85a.

Par90b.

Par88a.

Par90a.
D. Parkinson and J. Litt, *Massively Parallel Computing with the DAP*, MIT Press,

Par89a.

Par90c.

Per86a.

Pet91a.

Pog88a.

Ref93a.

Ref92a.

Ref90a.

Ric89a.

Ros90a.

Rum87a.

Sab88a.

San92a.

Sej87a.

Sev89a.

Siv92a.

Sk90a.  

Sol91a.  

Ste90a.  

Supa.Supemode, Esprit Project P1085.

Tom88a.  

Tre89c.  

Tre89a.  

Tre82a.  

Tre90a.  

Tre89b.  

Tre92a.  

Ult89a.  

Val90a.  

Vel90a.  

Wer88a.  

Wit89a.  
Zen92a.  

Zha90a.  

Zha90b.  
APPENDIX 5.1 - Backpropagation in nC with the Necessary Extensions for Parallelisation

The text below presents the nC code of the build_rule procedure for backpropagation neural systems. The extensions provided to the original nC language are noted as "snode" in the code.

```c
#include <stdio.h>

int build_rules()
{
    int cn, i, j, k;
    int t;
    char buf[LINESIZE], buf1[LINESIZE];
    if (!sys || !sys->net || (O_LAYER < 0))
        return (FAIL);
    for (cn = 0; cn < sys->nets; cn++) {
        ** Initialize rules in the hidden and output layers **
        for (i=1; i<sys->net[cn]->layers; i++) {
            for (j=0; j<sys->net[cn]->layer[i]->clusters; j++) {
                for (k=0; k<sys->net[cn]->layer[i]->cluster[j]->neurons; k++) {
                    rule_init(
                        "neuron.state_upd",
                        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_state_upd],
                        &state_upd_class,
                        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->fanin,
                        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[N_STATE],
                        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[N_ACC],
                        EOP);
                }
            }
        }
        /* snode */
        /* knowledge => one value to send back: the state of the concerned neuron */
        sprintf(buf, "n_0_%d_0_*d0", i);
        keep_init(
            &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_state_upd],
            1,0, buf,
            EOP);
        keep_extend(
            &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_state_upd],
            0,
            &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[N_STATE],
            EOP);
        if (i == O_LAYER) {
            rule_init(
                "neuron.err_cal",
                &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
                &err_cal_output_class,
```
/* snode */
/* knowledge => need to send back the error and all weights of the concerned neuron */
sprintf(buf, n_0_2_0._d1++s_0_2_0._dA '');
keep_init(
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
    1+sys->net[cn]->layer[i]->cluster[j]->neuron[k]->synapses,
    0,
    buf,"null", EOP);
keep_extend(
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
    0,
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[N_ERR], EOP);
for( t=0 ; t < sys->net[cn]->layer[i]->cluster[j]->neuron[k]->synapses; t++)
    keep_extend(
        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
        t+1,
        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->synapse[t]->weight,
        EOP);
}
else {
    rule_init (
        "neuron.err_cal",
        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
        &err_cal_hidden_class,
        sys->net[cn]->layer[i+1]->cluster[j]->neurons,
        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[N_ERR],
        &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[N_STATE], EOP);
/* snode */
/* knowledge => nothing to send back; i.e the hidden error is needed only */
/* by the hidden neuron itself */
keep_init(
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
    0, 0, EOP);
}
rule_init (
    "neuron.weight_upd",
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_weight_upd],
    &weight_update_class,
    sys->net[cn]->layer[i-1]->cluster[j]->neurons,
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->state[N_ERR],
    &sys->net[cn]->parameters[NET_P_learn_rate].parameter,
    EOP);
/* snode */
/* knowledge => neuron are anyway sent when the output neuron has */
/* computed its error */
keep_init(
    &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[ NEU_R_weight_upd ],
    0, 0, EOP);
}
/* end for k ... */
}
/* end for j ... */
}
/* end for i ... */

/*** extending rules -- to fill in the parameter pointers ***/
for (i = 1; i <= O_LAYER; i++) {
    for (j = 0; j < sys->net[cn]->layer[i]->cluster[0]->neurons; j++) {
        for (k = 0; k < sys->net[cn]->layer[i-1]->cluster[0]->neurons; k++) {
            rule_extend (
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->rules[NEU_R_state_upd ],
                k,
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->synapse[k]->weight,
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->input_neuron[k]->state[ N_STATE ],
                EOP);
            
            rule_extend (
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->rules[NEU_R_weight_upd ],
                k,
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->synapse[k]->weight,
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->input_neuron[k]->state[ N_STATE ],
                EOP);
        }
    }
}
/* end for k ... */

/* snode */
/* knowledge => neuron state update needs to receive state of its input_neuron */
sprintf(buf, "n_%d_%d_0_Ad0", cn, i-1);
rule_template(
    &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->rules[NEU_R_state_upd ],
    0, buf,
    EOP);
/* snode */
/* knowledge => neuron weight update needs to receive learn_rate and state of its input_neuron */
sprintf(buf, "p_0d2++n_%d_%d_0_Ad0", cn, i-1);
rule_template(
    &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->rules[NEU_R_weight_upd ],
    1, buf,
    EOP);
}
/* end for j ... */

if (i != O_LAYER) {
    for (j = 0; j < sys->net[cn]->layer[i]->cluster[0]->neurons; j++) {
        for (k = 0, t = 0;
            k < sys->net[cn]->layer[i+1]->cluster[0]->neurons;
            k++, t = t+2) {
            rule_extend (
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->rules[NEU_R_err_cal ],
                k,
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->output_neuron[k]->state[ N_ERR ],
                &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->output_neuron[k]->state[ N_ERR ],
                164
&sys->net[cn]->layer[i]->cluster[0]->neuron[j]->output_neuron[k]->synapse[j]->wei
EOP ;
}
sprintf(buf,"n_0_2_0_Ad1+s_0_2_0_Ad**");
rule_template(
    &sys->net[cn]->layer[i]->cluster[0]->neuron[j]->rules[NEU_R_err_cal],
    0, buf,
    EOP ;
}
/* endif */
} /* end for i ... */

/*** Initialize the rules at the cluster level ****/
for (i=1; i<sys->net[cn]->layers; i++) {
for (j=0; j<sys->net[cn]->layer[i]->clusters; j++) {
    rule_init ( 
        "cluster.state_upd",
        &sys->net[cn]->layer[i]->cluster[j]->rules[ CLU_R_state_upd ],
        &state_upd_meta_class,
        sys->net[cn]->layer[i]->cluster[j]->neurons,
        EOP );
    /* snode */
    /* knowledge => state update each ground sends back its state */
    sprintf(buf, "gr_0_%d_0_Ad0 .n_0_%d_0_*d0", i, i);
    keep_init(
        &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_state_upd],
        0,
        0, buf, "null", EOP );
    sprintf(buf, "gr_0_%d_0_Ad0 .n_0_%d_0_Bd0", i, (i-1) );
    rule_template(
        &sys->net[cn]->layer[i]->cluster[j]->rules[ CLU_R_state_upd ],
        0, buf,
        EOP );
    rule_init ( 
        "cluster.err_cal",
        &sys->net[cn]->layer[i]->cluster[j]->rules[ CLU_R_err_cal ],
        &err_cal_meta_class,
        sys->net[cn]->layer[i]->cluster[j]->neurons,
        EOP );
    /* snode */
    /* however,put coze can be done automatically ref TEMPLATE.RULE */
    sprintf(buf, "gr_0_%d_0_Ad1" , i );
    if (i==1) {
        sprintf(buf, "gr_0_%d_0_Ad1 .n_0_2_0_Bd1+s_0_2_0_BdA", i ) ;
    /* sprintf(buf1, "n_0_2_0_B.1 s_0_2_0_B.0" ); */
    rule_template(
        &sys->net[cn]->layer[i]->cluster[j]->rules[ CLU_R_err_cal ],
        0, buf,
        EOP );
    }
else
{
    rule_template(
        &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_err_cal],
        0, buf,
        EOP);
    sprintf(buf, "gr_0_2_0_Adl,n_0_2_0_*dl++s_0_2_0_*dA");
    for (t=0; t < sys->net[cn]->layer[i]->cluster[j]->neurons; t++)
        keep_init(
            &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_err_cal],
            0,
            0,
            buf,"null",
            EOP);
    }
}
/* snode */
rule_template(
    &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_weight_upd],
    0, buf,
    EOP);
}

/*** Extending rules at the cluster level ***/
for (i=1; i<sys->net[cn]->layers; i++) {
    for (j=0; j<sys->net[cn]->layer[i]->clusters; j++) {
        for (k=0; k<sys->net[cn]->layer[i]->cluster[j]->neurons; k++) {
            rule_extend (  
                &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_state_upd],
                k,
                &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_state_upd],
                EOP);
            rule_extend (  
                &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_err_cal],
                k,
                &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_err_cal],
                EOP);
            rule_extend (  
                &sys->net[cn]->layer[i]->cluster[j]->rules[CLU_R_weight_upd],
                k,
                &sys->net[cn]->layer[i]->cluster[j]->neuron[k]->rules[NEU_R_weight_upd],
                EOP);
        }
    }
}


```c
for (i=1; i<sys->net[cn]->layers; i++) {

    rule_init(
        "layer.state_upd",
        &sys->net[cn]->layer[i]->rules[ LAY_R_state_upd ],
        &state_upd_meta_class,
        sys->net[cn]->layer[i]->clusters,
        EOP);

    rule_init(
        "layer.err_cal",
        &sys->net[cn]->layer[i]->rules[ LAY_R_err_cal ],
        &err_cal_meta_class,
        sys->net[cn]->layer[i]->clusters,
        EOP);

    rule_init(
        "layer.weight_upd",
        &sys->net[cn]->layer[i]->rules[ LAY_R_weight_upd ],
        &weight_update_meta_class,
        sys->net[cn]->layer[i]->clusters,
        EOP);

    /* snode */
    /* knowledge => ref to RESTRICT which implies all meta that will form a controller */
    /* to indicate its para and keep list template */
    sprintf(buf, "p_0d2++n_0_%%d_0_Bd0" , (i-1) ) ;
    rule_template(
        &sys->net[cn]->layer[i]->rules[LAY_R_weight_upd],
        0, buf,
        EOP);
}

/*** Extending rules at the layer level ***/
for (i=1; i<sys->net[cn]->layers; i++) {
    for (j=0; j<sys->net[cn]->layer[i]->clusters; j++) {
        rule_extend(
            &sys->net[cn]->layer[i]->rules[ LAY_R_state_upd ],
            j,
            &sys->net[cn]->layer[i]->cluster[j]->rules[ CLU_R_state_upd ],
            EOP);

        rule_extend(
            &sys->net[cn]->layer[i]->rules[ LAY_R_err_cal ],
            j,
            &sys->net[cn]->layer[i]->cluster[j]->rules[ CLU_R_err_cal ],
            EOP);

        rule_extend(
            &sys->net[cn]->layer[i]->rules[ LAY_R_weight_upd ],
            j,
            &sys->net[cn]->layer[i]->cluster[j]->rules[ CLU_R_weight_upd ],
            EOP);
    }
}
```
/*** Initialize the rules at the net level ***/
/*** Initialize the recall rule ***/
rule_init (
    "net.recall", /* recall() takes layers-1 steps to */
    &sys->net[cn]->rules[ NET_R_recall ], /* update the states of all layers */
    &recall_meta_class, /* except the input layer */
    sys->net[cn]->layers-1,
    EOP);
/*** Initialize the tolerance rule ***/
/* tolerance is a ground rule which */
/* calculates the Euclidean distance, Hamming */
/* distance or the maximum error of the */
/* output pattern according to the control */
rule_init (
    "net.tol_test", /* control switch stored NET_P_measure */
    &sys->net[cn]->rules[ NET_R_tol_test ], /* result placed in NET_P_score */
    &tol_test_class,
    sys->net[cn]->fanout,
    &sys->net[cn]->parameters[ NET_P_tolerance ].parameter,
    &sys->net[cn]->parameters[ NET_P_score ].parameter,
    &sys->net[cn]->parameters[ NET_P_measure ].parameter,
    EOP);
/* snode incorporated ground rules n=> no para list template */
keep_init(
    &sys->net[cn]->rules[ NET_R_tol_test],
    0, 0, EOP);
sprintf(buf1, "" );
rule_template(
    &sys->net[cn]->rules[ NET_R_tol_test],
    0, buf1,
    EOP);
/* Initialize the weight update meta_rule at net level */
rule_init (
    "net.weight_upd", /* except the input layer */
    &sys->net[cn]->rules[ NET_R_weight_upd ],
    &weight_update_meta_class,
    sys->net[cn]->layers - 1,/* except the input layer */
    EOP);
sprintf(buf, "mr_0_1d2 ,, p_0d2++n_0_0_0_Bd0" );
sprintf(buf1, "mr_0_2d2 ,, p_0d2++n_0_1_0_Bd0" );
rule_template(
    &sys->net[cn]->rules[ NET_R_weight_upd],
    0, buf, buf1,
    EOP);
/*** Initialize the learn rule ***/
rule_init ( "net.learn",..."}
&sys->net[cn]->rules[ NET_R_learn ],
&learn_meta_class,
2 + 2*(sys->net[cn]->layers-1),
EOP);

/*** Extend the recall rule ***/
for (i=1; i<sys->net[cn]->layers; i++) {
    rule_extend (    
        &sys->net[cn]->rules[ NET_R_recall ],
        i - 1,
        &sys->net[cn]->layer[i]-rules[ LAY_R_state_upd ],
        EOP);
}

/*** Extend the tolerance rule ***/
for (i=0, k=0; i<sys->net[cn]->layer[0_LAYER]->clusters; i++) {
    for (j=0; j<sys->net[cn]->layer[0_LAYER]->cluster[i]->neurons; j++) {
        rule_extend (    
            &sys->net[cn]->rules[ NET_R_toI_test ],
            k++,
            &sys->net[cn]->layer[0_LAYER]->cluster[i]->neuron[j]->state[ N_TARGET ],
            &sys->net[cn]->layer[0_LAYER]->cluster[i]->neuron[j]->state[ N_STATE ],
            EOP);
    }
}

/* Extend the weight update rule at the net level */
for (i=1; i<sys->net[cn]->layers; i++) {
    rule_extend (    
        &sys->net[cn]->rules[ NET_R_weight_upd ],
        i - 1,
        &sys->net[cn]->layer[i]->rules[ LAY_R_weight_upd ],
        EOP);
}

/*** Extend the rule learn ***/
k=0;
for (i=1 ; i < sys->net[cn]->layers ; i++) {
    rule_extend (    
        &sys->net[cn]->rules[ NET_R_learn ],
        k++,
        &sys->net[cn]->layer[i]->rules[ LAY_R_state_upd ],
        EOP);
} ;

rule_extend (    
    &sys->net[cn]->rules[ NET_R_learn ],
    k++,
    &sys->net[cn]->rules[ NET_R_tol_test ], /* calculate the maximum error of the output units */
    EOP);
for (i=sys->net[cn]->layers-1; i>0; i--) {
    rule_extend (    
        &sys->net[cn]->rules[ NET_R_learn ],
        k++,
        &sys->net[cn]->rules[ NET_R_tol_test ], /* calculate the maximum error of the output units */
        EOP);
} ;

k=0;
for (i=1 ; i < sys->net[cn]->layers ; i++) {
    rule_extend (    
        &sys->net[cn]->rules[ NET_R_learn ],
        k++,
        &sys->net[cn]->rules[ NET_R_toI_test ], /* calculate the maximum error of the output units */
        EOP);
} ;
&sys->net[\text{cn}]->rules[\text{NET}_R\_learn],
\text{k}++,
&sys->net[\text{cn}]->layer[i]->rules[\text{LAY}_R\_err\_cal],/* calculate the errors */
EOP);
}

rule\_extend ( 
&sys->net[\text{cn}]->rules[\text{NET}_R\_learn],
\text{k}++,
&sys->net[\text{cn}]->rules[\text{NET}_R\_weight\_upd],
EOP);
};/* end for \text{cn} ... */
return (\text{OK});
APPENDIX 5.2 - Automatic Generation of TaskID -

The text below presents the C code which generates automatically the taskID array.

```c
/*------------------------------------------------------
/* SNODE : to analyse the Taskid */
/*.-----------------------------------------------------*/
#include <time.h>
long clock ;
#include "keep_sysdef.h"
extern system_type *sys ;
/* GLOBAL variable coming from hlc_produce */
extern int nongen_total ;
/* which Transputer */
int j =0 ;
/* which generic type */
int i=0;
/* total number of generic type */
int t =0 ;
/* total number of instance per generic type */
int *inst ;
/* max of the different total number of instance per generic type */
int max = 0 ;
/* the taskId association"array" layer, cluster, neuron */
/* WARNING no more than 25 layer, 10 cluster */
int *gen[25][10] ;
int **geny ;
/* which instance of generic */
int k=0 ;
/* the DIV (/) 48 and its sum */
int *a , suma=0 ;
/* the MOD (%) 48 and its sum(bDIVj) > 0 */
int *b ; sumb1_complex=0, sumb2_complex=0 ;
/* ---------------------------------------------i-------------------*/
TaskidProduceO
{
int 1, m,n, h, blip, x, z ;
int numb =0 ;
FILE *fdfile ;
FILE *fdesti ;
int cycles=0 ;
char name[50] ;
clock = time ( (long *) 0 ) ;
/* open (create) the ncsnode_config.h file */
if (! ( fdfile= fopen ( "ControlFile/TaskId" , "w" )) )
{
    printf ( "unable to open file0);
    return ( 0 );
}
fprintf(fdfile, "/* ncsnode_config.h produced at %s */ 0 , ctime ( &clock ));
```

171
/* open (create) the fill_desti file */
if ( ! ( fdesti = fopen ( "ControlFiIe/filI_desti.c", "w" ) ) )
{
    printf ( "unable to open fileO );
    return ( 0 );
};
fprintf(fdesti, "/* fill destination array produced at %s */\n", ctime ( &clock ) );
/* find the number of generic type ie nbe cluster per layer */
for (l=1 ; l < sys->net[0]->layers ; l++)
    for (n=0 ; n < sys->net[0]->layer[l]->clusters ; n++)
    {
        if (sys->net[0]->layer[l]->cluster[n]->neurons) t++ ;
        if (max < sys->net[0]->layer[l]->cluster[n]->neurons)
            max = sys->net[0]->layer[l]->cluster[n]->neurons ;
    };
/* allocate the a,b, inst, and gen array */
a = (int *)malloc(sizeof(int)* t) ;
b = (int *)malloc(sizeof(int)* t) ;
inst = (int *)malloc(sizeof(int)* t) ;
geny = (int **)malloc(sizeof(int *)* t ) ;
/* declare the desti_gen for SNODE */
for (l=1 ; l < sys->net[0]->layers ; l++)
{
    numb = 0 ;
    for (n=0 ; n < sys->net[0]->layer[l]->clusters ; n++)
        numb = numb + sys->net[0]->layer[l]->cluster[n]->neurons ;
    fprintf(fdesti, "int desti_gen_%d_leng= %d ; 0, 1, numb ) ;
    fprintf(fdesti, "int desti_gen_%d[%d] ; 0, 1, numb) ;
};
/* set number of instance per generic */
m=0 ;
for (l=1 ; l < sys->net[0]->layers ; l++)
    for (n=0 ; n < sys->net[0]->layer[l]->clusters ; n++)
    {
        inst[m] = sys->net[0]->layer[l]->cluster[n]->neurons ;
        geny[m] = (int *)malloc(sizeof(int )* inst[m]) ;
        gen[l][n] = (int *)malloc(sizeof(int) *  inst[m]) ;
        for (h=0 ; h < inst[m] ; h++)
            gen[l][n][m] = -1 ;
        m++ ;
    };
/* assign all the A and B */
for (l = 0 ; l < t ; l++)
    {  a[l] = inst[l] /48 ;
        b[l] = inst[l] % 48 ; 
    };
/* find the sum needed later */
for (l = 0 ; l < t ; l++)
    {  suma = suma + a[l] ;
    };
/* find the TaskId for each instance of each generic on a transputer */
j = 0 ;
k = 0 ;
/* k = 0 => Tj : taskld(gi_k) = i-1 */
for (i = 0; i < t; i++)
/* I counted without the (t)controller tasks => change */
geny[i][k] = (i+1) - 1 + nongen_total;

/* k > 0 => Tj : taskId(gi_k) */
/* = taskId( gi_(k-1) ) + SUM k <= i a[k] + b[k]>=j + SUM k>i a[k] + b[k]>=j-1 */
for (i = 0; i < t; i++)
{
j = 1;
for (k = 1; k < inst[i] ; k++)
{
    sumb1_complex = 0; sumb2_complex = 0;
    for (n = 0; n < i; n++)
        if (j == 0)
            sumb1_complex = sumb1_complex + ( b[n] > j );
    for (n = i; n < t; n++)
        if (j > 0)
            sumb2_complex = sumb2_complex + ( b[n] > j-1 );
    if (j > 0) geny[i][k] = geny[i][k-1] + suma + sumb1_complex + sumb2_complex;
    if (!j) geny[i][k] = cycles + geny[i][0];
    j = j + 1;
    if (j / 48) { j = 0; cycles++ ; } ; }
}
/* now fill the desti_gen arrays */
m=0;
fprintf(fdesti, "void FillDesti( 0) ;
fprintf(fdesti, "(0) ;
for (l=1 ; l < sys->net[0]->layers ; l++)
for (n=0 ; n < sys->net[0]->layer[l]->clusters; n++)
{
    for (k=0 ; k < inst[m] ; k++)
        if (n!=0 & & k ==0) z = sys->net[0]->layer[l]->cluster[n-1]->neurons;
        if (n==0 & & k==0) z = 0;
        fprintf(fdesti, " desti_gen_%d[%d] = %d ; 0 ,1 ,z ,geny[m][k]) ;
        z++ ;
    m++ ;}
fprintf(fdesti, ")0) ;
/* now allocate the gen array */
/*print */
m=0;
for (l=1 ; l < sys->net[0]->layers ; l++)
for (n=0 ; n < sys->net[0]->layer[l]->clusters; n++)
{
    for (k=0 ; k < inst[m] ; k++)
        { gen[l][n][k] = geny[m][k] ; }
    m++ ;}
Print(fdfile);
fclose(fdfile);
*/
APPENDIX 5.3 - Transputer Network Topology: Hypercube.map -

The text below presents <.map> file Transputer network topology representing an hypercube topology. Each line specifies the transputer connection to establish, each transputer device is referred to as an integer. 0 represents the Root transputer, 1 to 48 represents the worker transputer devices.

0: 1 42
1: 2 4 5
2: 3 6 18
3: 4 7 19
4: 8 12
5: 6 8 13
6: 7 22
7: 8 23
8: 16
9: 10 12 13 34
10: 11 14 26
11: 12 15 27
12: 16
13: 14 16
14: 15 30
15: 16 31
16:
17: 18 20 21 33
18: 19 22
19: 20 23
20: 24 36
21: 22 24 37
22: 23
23: 24
24: 40
25: 26 28 29 41
26: 27 30
27: 28 31
28: 32 44
29: 30 32 45
30: 31
31: 32
32: 48
33: 34 36 37
34: 35 38
35: 36 39 43
36: 40
37: 38 40
38: 39 46
39: 40 47
40:
41: 42 44 45
42: 43 46
43: 44 47
44: 48
APPENDIX 5.4 - Automatic Generation of Tasks Location

The text below presents the C code which generates automatically the file.hlc which declares and specifies the location of the 3L parallel Tiny tasks onto the SN1000.

/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/

void Locate(fghlc, control_total)
FILE *fghlc ;
int control_total ;
{
int l,m,n,x,blip, g, h,mm, hh, mmm, cc ;
int inst_total, total=0 , gen_total=0 ;
/* now try to create the Group and their placement */
int **location ;
int group_total=l ;
int counter, maximum ;

location = (int **)malloc(sizeof(int *)*t) ;
for (l=0 ; l < t ; l++) location[l] = (int *)malloc(sizeof(int )*48) ;
for (l=0 ; l < 48 ; l++) stop[l] = -1 ;
stop[0] = 1  ; n=l ;
/* assign location per type */
for (l=0 ; l < t ; l++)
{for(x=l; x <= 48 ; x++)
{ location[l][x] = a[l] ;
if (b[l] && b[l] >= x) { location[l][x]= location[l][x]+l ; }  ;
if(b[l] && b[l]+l ==x)
{  if ( 1  && mydiffer(x, stop) )
{ stop[n] = x; n++ ;group_total++; }  ;
if(!l) { stop[n] = x ; n++ ;  group_total++; }  ;
> ; /* don’t forget the last stop ie 48 if no empty tasks */
/* ie more than 48 neurons in one layer */
counter -= 0 ; maximum = 0 ;
for (l=1 ; l < sys->net[0]->layers ; l++)
{ counter -= 0 ;
for (n=0 ; n < sys->net[0]->layer[l]->clusters; n++)
{ counter = counter + sys->net[0]->layer[l]->cluster[n]->neurons ; }
if (maximum < counter) maximum = counter ;
};
if ( maximum > 48)
{ group_total++; stop[group_total-1] = 48 ; };
/* reorganise stop so to be ordered */
for (l=0 ; l < group_total ; l++)
{ for (n=1 ; n < group_total ; n++)
if (stop[l] != -1 && stop[n] != -1 && stop[l] > stop[n])
{ blip = stop[l] ; stop[l] = stop[n] ; stop[n] = blip ; } ; }
/* count the total number of generic */
for (l=1; l < sys->net[0]->layers; l++)
}
for (n=0; n < sys->net[0]->layer[l]->clusters; n++)
gen_total = gen_total + sys->net[0]->layer[l]->cluster[n]->neurons;

/* count the total nbe of tasks : 3 + nbe_controllers + nbe_generic + the empty one */
if (gen_total < 48)
{  total = 3 + control_total + gen_total ;
total = total + (48 - stop[group_total-l]) + 1 ;}
else
{ total = 3 + control_total + gen_total ; }

/* declare the group */
inst_total = 1 ;
for (g=0; g < group_total ; g++)
{ m=0 ; inst_total = 1 ; mm=0 ; hh=1, mmm=0 ;
  if (g) fprintf(fghlc, "Group autom_%d (0 , g) ;
  for (l=1 ; l < sys->net[0]->layers ; l++)
      for (n=0 ; n < sys->net[0]->layer[l]->clusters; n++)
      { if (g)
          { for(h=0 ; h < location[m][stop[g]-1] ; h++)
              { fprintf(fghlc, "Task Gen0%d%d : gen0%d%d_%d ; 0,l,n,l,n, (g+h) ) ;
                inst_total++; } ;
              m++; }
      else
          { for(h=0 ; h < location[m][1] ; h++)
              { fprintf(fghlc, "Task Gen0%d%d : gen0%d%d_%d ; 0,l,n,l,n,h ) ;
                inst_total++; } ;
              m++; } ;
  } ;
/* declare the instance of tiny */
fprintf(fghlc, '\n\nTask "') ;
if (g!=0)
WhichTiny(fghlc, inst_total ) ;
if (g==0)
WhichTiny(fghlc, (inst_total+control_total )) ;
fprintf(fghlc, ": tiny ; 0 ) ;
if (g!=0)
fprintf(fghlc, "tiny { in(%d, " , inst_total-l ) ;
if (g==0)
fprintf(fghlc, "tiny { in(%d, " , inst_total-l+control_total ) ;
for (l=1 ; l < sys->net[0]->layers ; l++)
    for (n=0 ; n < sys->net[0]->layer[l]->clusters; n++)
    { if (g)
        { for(h=0 ; h < location[mmm][stop[g]-1] ; h++)
            { fprintf(fghlc, "gen0%d%d_%d[2], ",l,n, (g+h)) ;
            mmm++; }
        else
            { if (control_total !)=0 && mmm==0) {
                for (cc =0 ; cc < control_total ; cc++)
                    fprintf(fghlc, "c0d%d[2], " , cc ) ;
                for(h=0 ; h < location[mmm][1] ; h++)
                    { fprintf(fghlc, "gen0%d%d_%d[2], ",l,n, (g+h)) ;

177
```c

```
fprintf(fghlc, "Empty{ in(*, *, tiny[1], taskid ), 0) ;
fprintf(fghlc, " out(*, *, tiny[1], type0 }  0) ;
fprintf(fghlc, " >  0) ;
*/ position the groups */
fprintf(fghlc, "Olace {  0 ) ;
fprintf(fghlc, "Root on [0] 0 ) ;
fprintf(fghlc, "autom_0 on [1] 0 ) ;
for (g=1 ; g < group_total ; g++)
( g != group_total - 1 )
( if (g!=1)
fprintf(fghlc, "autom_%d on [%d->%d] 0 , g, stop[g-1], stop[g-1] ) ;
if (g==1)
fprintf(fghlc, "autom_%d on [%d->%d] 0 , g, 2, stop[g-1] ) ;
else
( if (stop[g] == 48 )
fprintf(fghlc, "autom_%d on [%d->%d] 0 , g, stop[g-1], stop[g] ) ;
else
( if (stop[g] != 48 ) {
fprintf(fghlc, "autom_%d on [%d->%d] 0 , g, stop[g-1], stop[g-1] ) ;
fprintf(fghlc, "Theempty on [%d->%d] 0, stop[g] ) ;
fprintf(fghlc, " }  0 ) ;
*/ totally russian approach to change TOTAL for the group ROOT with the correct value */
fclose(fghlc) ;
TotalRussian(total) ;
}/*=============================================================*/
int mydiffer(theint, stop)
int theint;
int stop[] ;
{ int l,m,n ;
for (l=0 ; l < 48 ; l++) if (stop[l] == theint ) return(0) ;
return(1) ;
}/*=============================================================*/
Print(fd)
FILE *fd ;
{ int l,m,n ;
for (l=0 ; l < t ; l++)
( fprintf(fd, "Generic type %d 0,1+1 ) ;
for (m= 0 ; m < inst[l] ; m++)
( fprintf(fd, "gen%d_%d = %dO, l+l,m,geny[l][m] );};
fprintf(fd, "Genarray 0 ) ;
m=0 ;
for (l=1 ; l < sys->net[0]->layers ; l++)
for (n=0 ; n < sys->net[0]->layer[l]->clusters; n++)
( for (k=0 ; k < inst[m] ; k++)
( fprintf(fd, " gen[%d][%d][%d] = %dO , 1, n, k, gen[l][n][k]) ;
} ;
m++ ;
)}
WhichTiny(fdd, inte)
FILE *fdd;
int inte;
{
    switch(inte)
    {
        case 2 : fprintf(fdd,"SingleTiny") ; break ;
        case 3 : fprintf(fdd,"DoubleTiny") ; break ;
        case 4 : fprintf(fdd,"TripleTiny") ; break ;
        case 5 : fprintf(fdd,"QuadrupleTiny") ; break ;
        case 6 : fprintf(fdd,"QuintupleTiny") ; break ;
        case 7 : fprintf(fdd,"SextupleTiny") ; break ;
        default : fprintf(fdd,"UndefinedTiny "); break ;
    }
}
TotalRussian(number)
int number ;
{
    int i ;
    FILE *aawk ;
    char tempo[200], tempruss[30] ;
    /* keep the date */
    sprintf(tempo, "%s" , ctime(&clock)) ;
    strncpy(tempruss, tempo, 24) ;
    /* open (create) the AWK-HLC file */
    if ( !  ( aawk = fopen ( "ControlFile/Awking" , "w" ) ) )
    {
        printf ( "unable to open AWK-HLC fileO ");
        return ( 0 ) ;
    }
    /* search for TOTAL */
    fprintf(aawk, 
        "nawk '{gsub(/TOTAL/,}nawk '{gsub(/DATE/, 
    fclose(aawk) ;
    system("chmod +x ControlFile/Awking") ;
    system("ControlFile/Awking") ;
    system("rm ControlFile/temp") ;
    /* open (create) the AWK-DATE file */
    if ( !  ( aawk = fopen ( "ControlFile/AwkingDate" , "w" ) ) )
    {
        printf ( "unable to open AWK-DATE fileO ");
        return ( 0 ) ;
    }
    /* search for TOTAL */
    fprintf(aawk, 
        "nawk '{gsub/DATE/, 
    fclose(aawk) ;
    system("chmod +x ControlFile/AwkingDate") ;
    system("ControlFile/AwkingDate") ;
    system("cp ControlFile/temp2 ControlFile/thescript") ;
    system("rm ControlFile/temp2") ;
    system("rm ControlFile/autom.uncpt") ;
    }"
APPENDIX 5.5 - Application of Backpropagation: Created Files & Code -

This appendix gives the complete list of 3L Parallel C + Tiny files created by the mapping software, and the resulting files for execution onto the SN1000. It also describes representative part of the code.

/* CODE GENERATED */
/* Task declaration and allocation, automatically generated */
-rw-r-r-- 1 magali  4499 Apr 12 1991 autom.hlc

/* 3L Parallel C Tiny source code of the tasks,
* control_ refer to Controller tasks and gen_gr_ refer to generic Worker tasks
* Both indicate their path within the nC structure, i.e. network,layer,cluster */
-rw-r-r-- 1 magali  18950 Apr 12 1991 tcontrol_mr_0d0.c
-rw-r-r-- 1 magali  14356 Apr 12 1991 tcontrol_mr_0d1.c
-rw-r-r-- 1 magali  178 Dec  3 1990 tcontrol_mr_0d1.lnk
-rw-r-r-- 1 magali  8390 Apr 12 1991 control_mr_0_1d2.c
-rw-r-r-- 1 magali  179 Dec  3 1990 control_mr_0_1d2.lnk
-rw-r-r-- 1 magali  8390 Apr 12 1991 control_mr_0_2d2.c
-rw-r-r-- 1 magali  179 Dec  3 1990 control_mr_0_2d2.lnk
-rw-r-r-- 1 magali  1091 Dec  3 1990 generic_gr_0_1_0.h
-rw-r-r-- 1 magali  948 Dec  3 1990 generic_gr_0_2_0.h
-rw-r-r-- 1 magali  1345 Apr 12 1991 theground1.c
-rw-r-r-- 1 magali  1233 Apr 12 1991 theground2.c
-rw-r-r-- 1 magali  1022 Apr 12 1991 empty.c
-rw------- 1 magali   163 Dec  3 1990 empty.lnk
-rw-r-r-- 1 magali  3311 Apr 12 1991 fill_desti.c
-rw-r-r-- 1 magali  5949 Apr 12 1991 generic_gr_0_1_0.c
-rw-r-r-- 1 magali  179 Dec  3 1990 generic_gr_0_1_0.lnk
-rw-r-r-- 1 magali  5826 Apr 12 1991 generic_gr_0_2_0.c
-rw-r-r-- 1 magali  179 Dec  3 1990 generic_gr_0_2_0.lnk

/* Code of the global controller task: global.* which controls the Input Output operations with the host */
-rw-r-r-- 1 magali  27528 Apr 17 1991 global.c
-rw------- 1 magali    212 Dec  3 1990 global.h
-rw------- 1 magali    162 Dec  3 1990 global.lnk

/* Generic data declaration: the nC data structure, the Pygmalion reading and writing procedures */
-rw-r-r-- 1 magali    2154 Dec  3 1990 ncsnode.h
-rw-r-r-- 1 magali    891 Dec  3 1990 ncsnode_config.h
-rw-r-r-- 1 magali  8054 Apr 12 1991 snode_load.c
-rw-r-r-- 1 magali    3793 Apr 12 1991 write.c

/* the script for generating the tasks's declaration, allocation and code and
* for compiling through the 3L Parallel C + Tiny software */
-rwx------- 1 magali   1259 Dec  3 1990 thescript*

/* RESULTING EXECUTABLE CODE */
/* Compulsory files */
Task's Code: An Example

The text below presents the source code of a Controller task, and more particularly the control_mr_0_1d2.c.

#include <chan.h>
#include <tiny.h>
#include <stdlib.h>
#include "fill_desti.c"
/* Look Up table for generic 0 1 */
extern int desti_gen_l[24] ; extern int desti_gen_l_leng = 24 ;
/* Look Up table for generic 0 2 */
extern int desti_gen_2[96] ; extern int desti_gen_2_leng = 96 ;
/* Define MASK SIZE */
#define MASK_SIZE 2
/* declaration for controller koliko clusters ... */
int koliko_cluster[2] ;
int koliko_neuron[2][1] ;
/* Partial Data Declaration */
/* Buffer for data exchanges */
float *buffer ;
/* Various counter */
int i, j, k, t;
/* Various variable for data exchanges */
int destination, source, theid, clu;
/* Various variable for keeping creatins controller Ids */
int *ParFrom;
int ParFrom-lg;
/* Message Delivery system */
int typing;

/* Data Exchange Routines */
/*---------------------------------*/
/* Meta To Meta Data Exchanges */
/*---------------------------------*/
int MMsend_exec(to, order)
int to;
int order;
{
int orders[2];
/* printf(should send exec order to */
orders[0] = order; orders[1] = 1;
t_sseq(typing, to, orders, 2*sizeof(int));
}
/*---------------------------------*/
int MMwait_exec(from)
int from;
{
int capt[2], Isource, longueur;
/* printf( "should wait order
-1
from
%sO
, from) */
longueur = t_recv(typing, &lsource, capt, 2*sizeof(int));
if (lsource != from) return(0);
return(1);
}
/*---------------------------------*/
int MMsend_end(to)
int to;
{
int orders[2];
/* printf('should send end (=1) to */
orders[0] = 1; orders[1] = 1;
t_sseq(typing, to, orders, 2*sizeof(int));
}
/*---------------------------------*/
int MMwait_end(fromid_array, leng)
int fromid_array[];
int leng;
{
int i, longueur, twoints[2]; int lsouce;
/* should wait an int != 1 */
longueur = t_recv(typing, &lsouce, twoints, 2*sizeof(int));
/* check that lsouce if member of Parfrom */
for (i = 0; i < leng; i++)
{
if (lsouce != fromid_array[i]) return(1);
}
return(0);
}
/*---------------------------------*/
/* process(name, what) */
/* printf("process ground called %s param %s 0, name, what") */
/***********************/
int MMsend_data(dest_id, buff, leng)
int dest_id;
float buff[];
int leng;
{
/* send the buffer to dest_id */
t_sseq(typing, dest_id, buff, leng*sizeof(float));
return(1);
}
/*****************************/
int MMwait_data(from, buff, leng)
int from;
float buff[];
int leng;
{
int isource, longueur;
/* receive in buff some data from task id from */
longueur = t_recv(typing, &isource, buff, leng*sizeof(float));
if (longueur != leng*sizeof(float)) return(0);
if (isource != from) return(0);
return(1);
}
/*****************************/
/* Meta To Ground Data Exchanges
/*****************************/
int MGsend_data(dest_id, buff, leng)
int dest_id;
float buff[];
int leng;
{
/* send the buffer to the task with id dest_id */
t_sseq(typing, dest_id, buff, leng*sizeof(float));
return(1);
}
/*****************************/
int MGsend_exec(dest_id, order)
int dest_id;
int order;
{
int orders[2];
/* send the int order to the task with id dest_id */
orders[0] = order; orders[1] = order;
t_sseq(typing, dest_id, orders, 2*sizeof(int));
}
/*****************************/
int MGwait_data(name, buff, leng)
int name[];
float buff[];
int leng;
{
int isource, longueur;
/* receive in a buff the data from */
/* an id member of the name array */
/* send back the id received */
longueur = t_recv(typing, &isource, buff, leng*sizeof(float));
if (longueur != leng*sizeof(float))
    return(0);
return(isource);
}
/* MGpar_wait_end() */
int MGpar_wait_end()
{
    int Isource, longueur, ends[2];
    /* should recive the int (1) from */
    /* a member of desti_array_id */
    longueur = t_recv(typing, &lsource, ends, 2*sizeof(int));
    if (longueur != 2*sizeof(int)) return(0);
    if (ends[0] != 1) return(0);
    return(lsource);
}

/* TINY MAIN */
/*--------------------------------*/
main(argc, argv, envp, inv, outv, outc)
int argc, inc, outc;
char *argv[], *envp[];
CHAN *inv[], *outv[];
{
   CHAN *toKernel = outv[2];
    CHAN *fromKernel = inv[2];
    int taskId = (int)inv[3];

    int mask[MASK_SIZE];
    int initOk;
    int type;
    int twoints[2], result[2], width;

    /* Initialise the interface to Tiny */
    mask[0] = MASK_TYPE_ASYN;
    mask[1] = MASK_TYPE_SYNC;
    initOk = t_init(toKernel, fromKernel, taskId, MASK_SIZE, mask);
    /* testing the initialisation */
    if (initOk != 0)
    {
        /* fprintf(stdout, t_init() failed with value %d initOk); */
        return(-1);
    }
    /* type of message delivery to use: */
    /* async, sync, not read or not write */
    type = (int) outv[3];
    /* Assign to GLOBAL var */
    typing = type;
    /* call the procedure that fills correctly the look up tables*/
    FillDesti();
    koliko_cluster[0]=1;
    koliko_neuron[0][0] = 24;
    koliko_cluster[1]=1;
    koliko_neuron[1][0] = 96;
    for(;;)
    {
        mr_0_1d2();
    }
/* Meta wait an exec order from Meta called mr_0d2 */
MMwait_exec( 1 ) ;
/* MMwait_data */
/* wait tag mr_0d2 data p_0d2++n_0_0_0_Bd0 */
buffer = (float *)malloc(sizeof(float) * 97) ;
/* Meta wait data from Meta named mr_0d2 */
MMwait_data(1, buffer, 97) ;
if ( !mr_0_l_0d2() ) return(0) ;
MMsend_end( 1 ) ;
return(1) ;
;
int mr_0_l_0d2()
{
/* par_send( "gr_0_l_0_Ad2 ", "p_0d2++n_0_0_0_Bd0" ) ; */
t= 0 ;
for (i = 0; i < 24 ; i++)
{
    destination = desti_gen_l[t] ;
    MGsend_exec(destination, 2) ;
    t=t+1 ;
};
/* buffer has been allocated,send buffer to appropriate */
t= 0 ;
for (j = 0 ; j < 24 ; j++)
{
    destination = desti_gen_l[t] ;
    MGsend_data(destination, buffer, 97) ;
    t=t+1 ;
};
/* MGpar_wait_end( "gr_0_l_0_Ad2 ", "(null)" ) ; */
/* parallel wait end(gen_dest array, fromO to this value, end=l) */
for ( i = 0 ; i < 24 ; i++)
{
    theid = MGpar_wait_end() ;
    if ( NeuronAssoc(desti_gen_l, desti_gen_l_leng, theid, 1) == -1 )
        return(0) ;
}
return(1) ;
}
/*------------------------*/
APPENDIX 6.1 - DAP Code for the Matrix-matrix Mapping -

The text below presents the Fortran Plus code for a matrix-matrix mapping, for which two copies of the entire neural systems are held by the DAP. It is parameterised in order to accept different configurations, i.e., the neural system’s configuration consists of (X,Y,X).

C The code for a matrix-matrix decomposition scheme
C Timing one learning & one recalling

C calling dapent_ from the C host program
entry subroutine dapentry
C declaration of the common variables
common /givenblk/giv(*X,*10)
common /targetblk/tar(*X,*10)
common /zeroblk/ x(*X)
common /hidyblk/ y(*Y)
common /outzblk/ z(*X)
common /outtzblk/ tz(*X)
common /wghtOblk/ wO(*Y,*X)
common /wghtlblk/ wl(*X,*Y)
common /param/ p(3)
common /temps/ hiver(3)
C declaration of dap local variables
real xl(*X) real tzl(*X) real zl(*X)
real x2(*X) real tz2(*X) real z2(*X)
real x3(*X) real tz3(*X) real z3(*X)
real yall(*3Y) real eyall(*3Y)
real zall(*3,*X) real tzall(*3,*X)
real ezall(*3,*X) real ezl(*X) real ez2(*X) real ez3(*X)
real wOall(*3Y,*X) real deltwO(*3Y,*X) real accdeltwO(*3Y,*X)
real twO(*Y,*X) real ttwO(*Y,*X) real l0(*3Y,*X)
real w1all(*X,*3Y) real deltwt1(*X,*3Y) real accdeltwt1(*X,*3Y)
real tw1(*X,*Y) real ttw1(*X,*Y)
real l(*X,*3Y)
real tmpy(*3Y,*X)
logical M1(*3Y,*X) logical M11(*3Y,*X) logical tmpl(*Y,*X)
real tmpz(*X,*3Y)
logical M2(*X,*3Y) logical M21(*X,*3Y) logical tmp2(*X,*Y)
real tol integer numbpat integer count integer total
real maxi real mini
C declaration for timing
real*4 cput real*4 allt
external qmw_start_timing
external qmw_get_cpu_time
external qmw_get_elapsed_time
C receive the data from the host and convert them to the DAP format
call convhtod(x) call convhtod(giv) call convhtod(tar)
call convhtod(y)
call convhtod(z)
call convhtod(w0) call convhtod(w1)
call convhtod(p)
C DAP code starting, first initialise the local variables
M1 = .FALSE. tmp1 = .TRUE. M2 = .FALSE. tmp2 = .TRUE.
l = p(1) l0 = p(1)
zall = 0.0 ezall = 0.0
tol = p(2)
numbpat = p(3) / 3
total = 0
accdeltw0 = 0.0 accdeltw1 = 0.0
call set_mat(M1,1,1,tmp1) call set_mat(M11,1,1,tmp1) call set_mat(M11,(Y+1),1,tmp1)
call set_mat(M2,1,1,tmp2) call set_mat(M21,1,1,tmp2) call set_mat(M21,(Y+1),1,tmp2)
call SET_MAT(w0all,1,1,w0) call SET_MAT(w0all,(Y+1),1,w0) call SET_MAT(w0all,(2Y+1),1,w0)
call SET_MAT(w1all,1,1,w1) call SET_MAT(w1all,1,(Y+1),w1) call SET_MAT(w1all,1,(2Y+1),w1)

C this is IT (loop for 5 times accross the set)
call qmw_start_timing
do 600 kkk=l, Y0
  if (total.EQ.4) goto 900
count = 0
total = 0
C this is the running through each pattern at a time
do 400 kk=l, numbpat
C Get the training vector; i.e. input and target
  if (count.EQ.9) count = 7
  count = count + 1
  xl = giv(:,count) tzl = tar(:,count)
  count = count + 1
  x2 = giv(:,count) tz2 = tar(:,count)
  count = count + 1
  x3 = giv(:,count) tz3 = tar(:,count)
  tzall(1,) = tzl tzall(2,) = tz2 tzall(3,) = tz3
  accdeltw0 = 0.0 accdeltw1 = 0.0

C this is the processing loop
  do 200 k=l, Y00
    if (k.EQ.1) zl = 0.0 if (k.EQ.1) z2 = 0.0 if (k.EQ.1) z3 = 0.0
    if (k.EQ.1) zall = 0.0 if (k.EQ.1) yall = 0.0
    tmpy = MERGE(MATR(xl,3Y),MATR(x2,3Y),M1)
yall=SUMC(w0all* MERGE(tmpy,MATR(x3,3Y),M11))
yall = 1 / (1 + EXP(-yall))
    tmpz = w1all * MATR(yall,X)
zall(1,) = SUMC( GET_MAT(tmpz,1,1,X,Y) )
zall(2,) = SUMC( GET_MAT(tmpz,1,(Y+1),X,Y) )
zall(3,) = SUMC( GET_MAT(tmpz,1,(2Y+1),X,Y) )
zall = 1 / (1 + EXP(-zall))
    ezall = (tzall - zall) * zall * (1 - zall)
ezl = ezall(1,) ez2 = ezall(2,) ez3 = ezall(3,)
tmpz = MERGE(MATC(ezl,3Y),MATC(ez2,3Y),M2)
eyall = SUMR(w1all*MERGE(tmpz,MATC(ez3,3Y),M21))eyall = eyall * yall * (1 - yall)
deltw0 = 10*MATC(eyall,X)*MERGE(tmpy,MATR(x3,3Y),Ml1)accdeltw0 = accdeltw0 + deltw0deltwl = 1*MERGE(tmpz,MATC(ez3,3Y),M21)*MATR(yall,X)accdeltwl = accdeltwl + deltwl
w0all = w0all + deltw0 w1all = w1all + deltw1

maxi = MAXV(tzall -zall)
mini = MINV(tzall -zall)
if ( (-mini).GT.maxi) maxi = - mini

if ((maxi.LE.tol).AND.(K.LE.l)) total = total + 1
if (maxi.LE.tol) goto 400
200 continue
tw0 = get_mat(accdeltw0,1,1,Y,X)
ttw0 = get_mat(accdeltw0,(Y+1),1,Y,X)
w0 = w0 + tw0 + ttw0 + get_mat(accdeltw0,(2Y+1),1,Y,X)twl = get_mat(accdeltw1,1,1,X,Y)
ttwl = get_mat(accdeltw1,1,(Y+1),X,Y)
w1 = w1 + tw1 + ttwl + get_mat(accdeltw1,1,(2Y+1),X,Y)
call SET_MAT(w0all,1,1,w0) call SET_MAT(w0all,(Y+1),1,w0) call SET_MAT(w0all,(2Y+1),1,w0)call SET_MAT(w1all,1,1,w1) call SET_MAT(w1all,1,(Y+1),w1) call SET_MAT(w1all,1,(2Y+1),w1)
400 continue
600 continue
900 call qmw_get_cpu_time(cput)
call qmw_get_elapsed_time(allt)
pause 22
C Now the DAP has finished its work, send back some data
202 call convdtoh(x) call convdtoh(z)call convdtoh(w0) call convdtoh(w1)call convdtoh(tar)call convdtoh(hiver)
return
end
APPENDIX 6.2 - DAP Code for the Matrix-vector Mapping -

The text below presents the Fortran Plus code for a matrix-vector mapping. It is parameterised in order to accept different configurations: the neural system's configuration consists of (X,Y,X).

C June 92 matrix vector version
C Timing the whole learning cycle

entry subroutine dapentry

C declaration of the common variables
common /givenblk/giv(*X,*10)
common /targetblk/tar(*X,*10)
common /zeroblk/ x(*X)
common /hidyblk/ y(*Y)
common /outzblk/ z(*X)
common /outtzblk/ tz(*X)
common /wght0blk/ w0(*Y,*X)
common /wght1blk/ w1(*X,*Y)
common /param/ p(3)
common /temps/ hiver(3)

C declaration of dap local variables
real ey(*Y) real ez(*X)
real yone(*Y) real zone(*X)
real lr0(*Y,*X) real lr1(*X,*Y)
real tol
integer numbpat integer count integer total
real temp real maxi real mini

C declaration for timing
real*4 cput real*4 allt
external qmw_start_timing
external qmw_get_cpu_time
external qmw_get_elapsed_time

C receive the data from the host and convert them to the DAP format
call convhtod(x) call convhtod(giv) call convhtod(tar)
call convhtod(y)
call convhtod(z)
call convhtod(w0) call convhtod(w1)
call convhtod(p)

C DAP code starting, first initialise the local variables
lr0 = p(1) lr1 = p(1)
z = 0.0 ez = 0.0 y = 0.0 ey = 0.0
tol = p(2)
numbpat = p(3)
yone = 1.0 zone = 1.0 total = 0
call qmw_start_timing

C this is IT (loop for 500 times accross each pattern)
do 600 kkk=1, 500
if (total.EQ.10) goto 900
total = 0

C this is the running through each pattern at a time
do 400 kk=1, numbpat
C Get the training vector; i.e. input and target
    x = giv(kk)
    tz = tar(kk)
C this is the processing loop
    do 200 k=1, 5000
        if (k.EQ.1) y = 0.0 if (k.EQ.1) z = 0.0
        y = SUMC(w0 * MATR(x,Y))
        y = yone / (yone + EXP(-y))
        z = SUMC(w1 * MATR(y,X))
        z = zone / (zone + EXP(-z))
        ez = (tz - z) * z * (1 - z)
        ey = SUMR(w1 * MATC(ez,Y))
        ey = ey * y * (1-y)
        w0 = w0 + lr0 * MATC(ey,X) * MATR(x,Y)
        w1 = w1 + lr1 * MATC(ez,Y) * MATR(y,X)
        maxi = MAXV(tz - z)
        mini = MINV(tz - z)
        if ( (-mini).GT.maxi) maxi = - mini
        if ((maxi.LE.tol).AND.(K.LE.1)) total = total + 1
        if (maxi.LE.tol) goto 400
    200 continue
    400 continue
    600 continue
900 call qmw_get_cpu_time(cput)
call qmw_get_elapsed_time(allt)
pause 22
C Now the DAP has finished its work, send back some data
202 call convdtoh(x) call convdtoh(z)
call convdtoh(w0) call convdtoh(w1)
call convdtoh(tar)
call convdtoh(hiver)
return
end
APPENDIX 7.1 - Prototype of ASN.1 Macros for Defining the Space of Neural Models -

The text below presents a prototype of ASN.1 Macros for defining the space of neural models.

```
NNMODEL MACRO ::= BEGIN
  TYPENOTATION ::= Networks (Structure, CompCalculation)
  VALUENOTATION ::= Networks ::= Networks | Networks Network
  Network ::= "NETWORK" Data Groups," "," NetworkComp | empty
  Group ::= Slabs | NeuronsConnections
  Slabs ::= Slabs | Slabs Slab
  Slab ::= "SLAB" SlabSub," "," SlabData "," SlabComp | empty
  SlabSub ::= NeuronsConnections
  NeuronsConnections ::= Neurons "," Connections
  Neurons ::= "NEURON" NeuronData "," NeuronComp | empty
  Connections ::= "CONNECTION" ConnectionData "," ConnectionComp | empty
  NetworkData ::= EXTERNAL
  -- and similarly for all subordinates;
  -- i.e. groups, slabs , neurons and connection
  NetworkComp ::= NetworkComp | Comp "," NetworkComp
  -- and similarly for SlabComp and NeuronComp and ConnectionComp
  -- SlabComp ::= SlabComp | SlabComp "," SlabComp
  -- NeuronComp ::= NeuronComp | NeuronComp "," NeuronComp
  -- ConnectionComp ::= ConnectionComp | ConnectionComp "," ConnectionComp
  Comp ::= ParallelComp | SequentialComp | IndifferentComp
  ParallelComp ::= "par" CompParameters "," CompCalculation
  SequentialComp ::= "seq" CompParameters "," CompCalculation
  IndifferentComp ::= "na" CompParameters "," CompCalculation
  CompParameters ::= FunctionApplied ParametersApplied
```
-- the ParametersApplied apply to the subordinate level
-- specify level and data addressed

FunctionApplied ::= "F" printableString | "f" printableString
-- F applies to non neuron and connection levels
-- f applies solely to neuron and connection

END