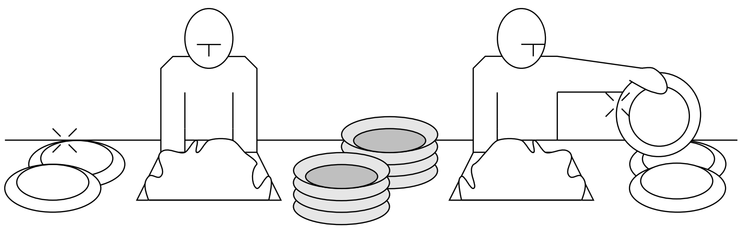
THE P IN PREENS

A dataset parallel interface for convis
and experiences with a parallel
counterpropagation simulator

Eric Boon



thesis 415
University of Nijmegen
Dept. Computer Science

This document is prepared with $\LaTeX$

Preface

Work expands as to fill the time available for its completion.
— One of Murphy's Laws

Murphy was an optimist.
— Unknown

When I first stepped into Theo Schouten's room and informed about graduation projects on neural networking and parallel processing, I didn't have the slightest idea what I had brought about. Theo dragged Louis Vuurpijl into the room and the two of them began talking about neural networks, the Nijmegen SuperCluster, dataset parallellization, PREENS and CONVIS. Finally they turned to me and asked: "Does that sound interesting?"

"Uhhh, I guess so", I murmured, and so I started on the project which eventually led to the thesis you are now reading. I'm not quite sure whether the result is exactly what they had in mind in the beginning, but I'm content with it.

In the period of time that passed by as I was working on this project, I got help from a lot of people. Some of them really helped me a lot, some just gave me a gentle push in the right direction—I thank all of them. It's impossible to give a complete list, but some of them deserve a little extra attention.

I would first like to thank my parents for their endless patience. Despite all, they kept supporting me with all means—I guess that's because they're my parents. Further, I'd like to thank Theo Schouten for reading and approving my thesis, even after long periods of silence from my side.

I thank all of my friends for their moral support. Special thanks go to: Martin Krooshof (and Natascha Lieffers, of course), Manuel Bilderbeek, Jeroen Melchiors, Erwin Hebing, Frodo Looijaard and Matthew vd Woerdt. I guess they know why...

Eric Boon
June 18, 1997

Contents

Introduction

At the KUN, a general purpose neural network simulation environment is being built, called PREENS. PREENS consists of a front-end for the user, called CONVIS a large set of datastructures and routines and several neural network simulation implementations. CONVIS provides the user interface as a program running seperately from the actual neural network simulation. To be able to control a simulation program via CONVIS, the simulation program should be built using an interface, which is specified by PREENS.

The existing interface is focused purely on sequential neural network simulation programs. A small set of routines was implemented to be able to interface CONVIS with parallel simulation programs, but these were not really part of the interface and it was a tedious job to integrate an existing parallel neural network simulation with CONVIS, especially with respect to distributed datastructures.

Goal

The goal of this project is the design and implementation of an interface for CONVIS, that supports the distribution of datastructures of a neural network simulation program over a parallel computer. The interface has to comply to three constraints:

- The *user* of PREENS should not be bothered by the fact that he is running the simulation on a parallel computer. From his point of view, the interface should be completely transparent.

- The *programmer* should be able to easily integrate his existing (sequential) program with the parallel interface. For the programmer the interface should also be as transparent as possible.

- The parallel interface should not be restricted to one or two parallel computer configurations or data distribution methods; it must be as flexible as possible.

As hardware platform the Nijmegen SuperCluster transputer system will be used, but the parallel interface should be easy to port to other parallel architectures. The interface will be tested on a counterpropagation network simulation program.

Layout of the thesis

Chapter 1 introduces the reader into the world of neural networking. A very short description of a general neuron model is given and the most important operations found in neural network simulation environments.

Chapter 2 shows how data can be distributed on a parallel computer to achieve dataset parallellization. Two entry points are found to implement dataset parallellization in a neural network simulation program: the network data and the input set.

Chapter 3 contains the history and peculiarities of PREENS and CONVIS. The concepts on which CONVIS is based are described and an explanation on the communication between convis and the neural network simulation program is given.

Chapter 4 is dedicated to the hardware the dataset parallel interface was implemented on: the Nijmegen SuperCluster. This chapter explains how the transputer system is built up, how to configure it for a given topology and finally how to run an application on it.

Chapter 5 presents the design and implementation of the parallel interface. The extensions and adaptions that were made to the existing interface are described.

Chapter 6 describes the neural network simulation program the interface was tested on. The counterpropagation network was chosen as test case for several reasons. Main reasons were that this type of neural network was not part of the PREENS distribution, yet, and that the counterpropagation network was said to train relatively fast.

Chapter 7 will present the results obtained with the implementation of the interface and contains some conclusions on both CONVIS and the interface.

Chapter 1

Neural Networks

At first, some of the mysteries that neural networks and neural network simulations seem to be clouded in, will be unraveled. This chapter gives a very coarse and brief introduction into neural networking.

1.1 A Bit Of History

It is not easy to put a foot on the exact time of genesis of that part of scientific research that is concerned with neural networks and neural network simulations. Mankind has long since wondered how the process of thinking works and, as soon as man had discovered the center of thinking, how the human brain operates.

Despite this long tradition, the starting point of modern neural network research is normally laid by the paper McCullogh and Pitts wrote in 1943 [McC43]. In this paper they described how a network of very simple processing elements can perform any arithmetic or logical function. Six years later, Donald Hebb specified a general learning rule for the synapses of animal neurons [Heb49].

Combining these results, Rosenblatt and his collaborators built the first successful neurocomputer, MARK I. The results were promising and neural network research boomed, until Minsky and Papert published an article which showed the capabilities, but most and forall, the *in*capabilities of the neuron and neural network models used [Min69]. The argumentation they used looked that sound, that the research of neural networks declined and a period of almost ear straining silence started.

Around 1974, Werbos—amongst others—(re)discovered a method for training neural networks with more than one or two layers. This blew away enough of the argumentation used by Minsky and Papert for the incapabilities of neural networks for the neural network 'renaissance' to start. Nowadays, many variations of neural network models and neural network simulations exist and with the increasing capacity of modern computers, neural networks can be efficiently simulated.

1.2 The Biological Example

The ultimate example of a biological neural network is, of course, the human neural system. It consists of about 10^{11} neurons and 10^{15} connections [Hec90]. The neurons differ in physical appearance, according to their function; the receptive areas—dendrites—of an association unit somewhere in the brain are much larger and look more bushy than the dendrites of a simple sensor neuron in the spinal cord. All neurons, however, act in the same way. An example of a biological neuron is given in figure 1.1.

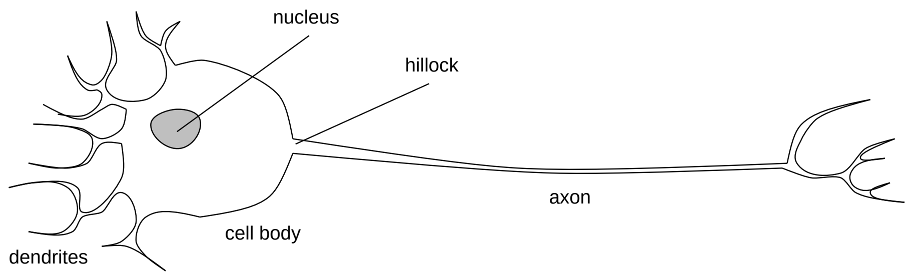


Figure 1.1: A biological neuron

A neuron can be divided in mainly three parts:

- dendrites
- cell body
- axon

The dendrites accept signals from other neurons. A neuron can have many dendrites. The signals accepted can be prohibitory as well as inhibitory and can have more or less impact on the total input of the neuron. Signals received by the dendrites are propagated to the cell body via the cell membrane. At the point on the cell body where the axon starts, the *hillock*, the input signals are 'accumulated'. When the signal level reaches a certain threshold, an outgoing signal is generated that runs through the axon—the neuron *fires*. A neuron has only one axon, but it's axon can have a forked end, where the fire-signal can be passed to the dendrites of other neurons [Was89].

1.3 Simulating A Neuron

With the knowledge presented above, a mathematical model of a 'general purpose' neuron can be derived. It is shown in figure 1.2. The dendrites have become input signals, whose effect on the total input is biased by a weight. This weight can be positive, for a prohibitory input, or negative for a inhibitory input. The inputs are summed, and an output signal is generated when the input sum reaches a certain threshold [Was89]. The input sum is called the neuron's *net input*, the output is called the *activation*.

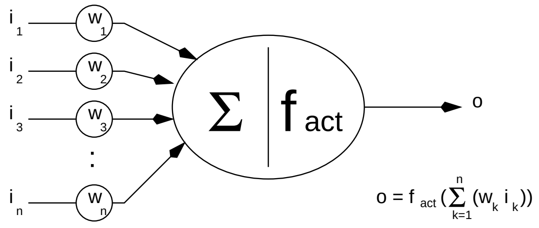


Figure 1.2: A mathematical model of a biological neuron

Often, an *activation function* is chosen instead of a simple threshold value. Sigmoid functions are very popular as activation function.

This is the most basic neuron model used in neural network simulations. Very often a more complex neuron is modeled, but the basis will always resemble the neuron in figure 1.2.

1.4 Building A Network

One single neuron can be an interesting object of study, but neurons become really useful when put together in a network. As said in the previous section, neurons accept incoming signals from other neurons and pass their output signal to others. A neural network is built by making connections between neurons, defining which neurons accept signals from which other neurons.

In a network, clusters of neurons can be found, where each cluster of neurons performs a particular task. Such clusters are called *layers* or *slabs*. Signals in a neural network are typically processed by more or less fixed sequence of layers. Two of these layers are of particular interest: the *input layer* and the *output layer*.

Neurons in the input layer do not accept incoming signals from other neurons; their activations are determined externally, by the user of the neural network. You can think of them as sensory neurons. Neurons in the output layer do not present their output to other neurons, but to the outer world—the user of the network. In figure 1.3 a simple neural network is shown with three layers; one input layer, one output layer and a layer in between, called a *hidden layer*. This network is called a feed forward network, since all connections progress the signal forward, away from the input neurons, towards the output neurons.

There is a general disagreement on how to count the layers. The input layer is often not regarded as a layer in the neural network, as it doesn't perform any calculation, but merely passes the input values to other neurons. In this view, the network in figure 1.3 is a two layer neural network.

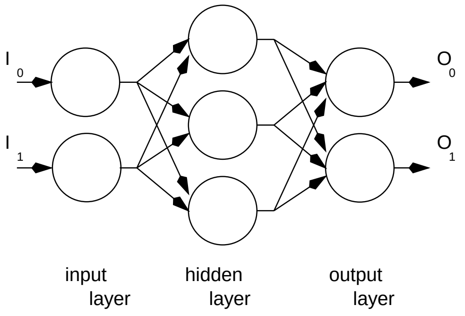


Figure 1.3: A simple neural network.

1.5 Training

After an initial network is built, it must be trained to perform a certain task. Training a neural network means setting and adjusting weights, or adding new neurons to the network. A for the task suitable set of training patterns must be determined. Training is then performed by presenting a pattern from the training set to the network and calculating the net activation. Then, some kind of *net error* is determined and adjustments to the network are calculated, that decrease the net error. These adjustments are then applied to the network and another pattern from the training set is selected and presented to the network.

These *training steps* are continued until all the patterns of the training set have been processed by the neural network. One such run of training steps is called an *epoch*. In most neural network paradigms, one epoch is not sufficient to train the network, thus epochs are repeated, until the net error does not exceed a previously set *error criterium* anymore.

1.5.1 Supervised And Unsupervised Training

Two ways exist to derive the net error and these two methods lay down two major classes of neural network simulations. When an input pattern is presented to the network and the net activation has been calculated, the net output can be compared with a well known target output that belongs to this output pattern. The net error can then be determined according to the differences found between the calculated net output and the desired net output, the target pattern. This is called *supervised learning*; a supervisor directs the adjustments made to the neural network according to the target patterns.

This kind of training is well suited for cases in which the input patterns can be divided in separate, pre-defined classes. The target patterns then represent one such class each.

The second method is *unsupervised learning*. The neural network simulation has associated with it an *energy function*, which indicates the 'level of activation' within the network. When an input pattern is presented to the network and the net activation is calculated, the energy function determines which neurons or weights in the network need to be adjusted. There is no indication what the proper output of the neural network should be when a certain input pattern is presented. The goal of training is to minimize the energy in the network.

This method can be of use when the classes of input patterns are not known at fore hand. The network determines these classes during the training process on basis of the energy function.

Of course, training paradigmas exist which comprise a mixture of these methods. The counterpropagation network discussed in chapter 6 is an example of such a hybride paradigma. The (hidden) Kohonen layer is trained using a kind of energy function, while the (output) Grossberg layer is trained supervised.

1.5.2 Training Per Pattern And Per Epoch

Independent from the training method used, supervised or unsupervised, the network is said to be well trained, when the net error drops below a preset threshold value. For supervised training, this means that when any of the input patterns is presented to the trained network, the output pattern will closely resemble the target pattern belonging to that input pattern. For unsupervised training, it means that the net energy will remain minimal in all cases. The algorithm below gives an general implementation of the training process described:

```
WHILE NOT well trained
REPEAT
    FOR EACH pattern IN training set
    REPEAT
        calculate net activation(pattern)
        calculate net error
        calculate network updates
        update network
    END REP
END REP
```

Note that this algorithm does not include how the net error is derived or how the network is adjusted; this algorithm holds for both supervised and unsupervised training processes. The real training of the network lies in the implementation of calculate net error and update network. Numerous training paradimas exist, among which *backpropagation* [Rum86] and *Kohonens self organizing map* [Koh84] are the most widely known.

The training algorithm presented above adjusts the network immediately when a pattern is presented to the network. This is known as training per pattern. Another method is to present the patterns in the training set to the network and calculate the required changes, but, instead of applying these changes immediately to the network, accumulate them and apply these accumulated change to the network.

```
WHILE NOT well trained
REPEAT
    FOR EACH pattern IN training set
    REPEAT
        calculate net activation(pattern)
        calculate net error
        calculate network updates
        accumulate network updates
    END REP
    update network
END REP
```

In this way the network is not updated with each pattern, but at the end of each epoch. The advantage of this method is that the network changes more graduately and can, possibly, be trained faster.

1.6 Recalling And Classifying

When the network is sufficiently trained, it is ready for normal usage. In most cases, the network will be tested first with a special set of test patterns. This testing is called the *recall* phase. Input patterns are presented to the network and the produced output is compared with the desired result. Recalling is very similar to training, but the network is not altered anymore. If the neural network doesn't produce satisfying results during the recall phase, it must be trained further, possibly with an adapted training patterns set.

Finally, the well trained and tested neural network can be used to operate on the data it was trained for. This final phase is called the *classifying* phase, because the neural network can now recognize the presented input patterns, match them with the trained input patterns and produce the best matching output pattern.

Chapter 2

Parallelization Of Neural Networks

Parallel processing can be defined as 'information processing that emphasizes on the concurrent manipulation of data elements belonging to one or more processes, solving a single problem' [Qui94]. This chapter will give an introduction in parallelization and describe how to introduce dataset parallelization in a sequential neural network simulation program.

2.1 Parallelism In A Program

A program consists of data structures and functions that operate on the data. This concept was once stated brief and to the point by Niklaus Wirth in the expression

algorithms + datastructures = programs [Wir76].

The search for a way to parallelize an existing sequential program can therefore start at taking a closer look at either the algorithms or the data. Taking the algorithms as starting point, will lead to a kind of parallelism that is called *control parallelism*. When the data structures are chosen, the result will be a program with *dataset parallelism*.

Take for example the process of doing the dishes. The algorithms in this 'program' obviously include washing up and wiping dry, while the data consists of the dirty dishes. When you're alone, you will first wash up all the dishes and then wipe it all dry. This is the sequential version of the doing-the-dishes program. When you're with two, one can wash up and the other can wipe dry. This is an example of control parallelism—the overall algorithm is split up into two minor tasks that can be performed more or less simultaneously. For dataset parallelism, each would take a half of the dishes and wash up and wipe it dry independently from the other. You would, of course, need two kitchen sinks in this case. (See fig. 2.1)

The rest of this chapter will only describe dataset parallelism for neural network simulation programs, as the interface for CONVIS was designed for dataset parallel programs. The two major datastructures found in a neural

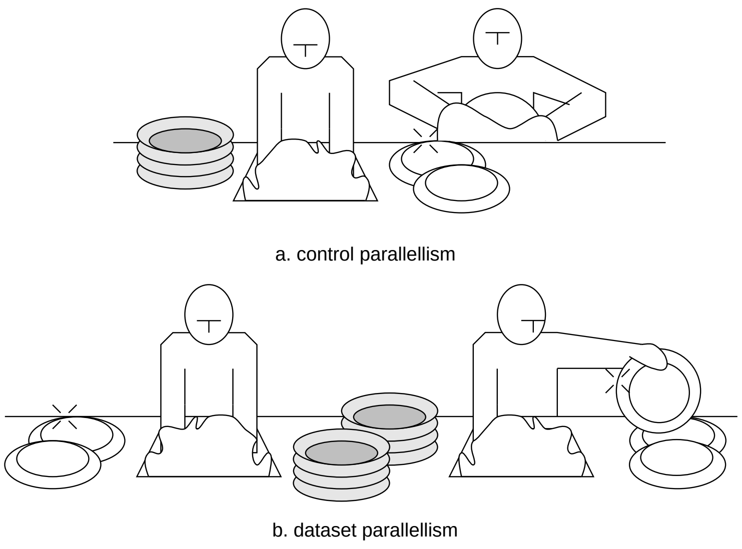


Figure 2.1: Doing the dishes as a parallel process

network simulation program are the network data and the patterns used for learning, recalling and classifying. For both these datastructures, a dataset parallelizing technique can be defined.

2.2 Splitting The Input

The inputset of a neural network simulation program usually consists of the several patternsets: training patterns, test patterns and the classification patterns. For inputset parallelization, the elements of the inputset are divided in several smaller subsets of patterns. On each processor in the parallel computer, a copy of the neural network simulation program is run, which each takes a different subset of the data and process that. This form of parallelization is normally useful, when a large set of input patterns is to be processed, or if the processing of individual patterns, eg. during training, is time-consuming.

Figure 2.2 shows an inputset parallel neural network simulation program. Each processor has its own copy of the neural network and a part of the complete inputset (in this case, the patterns A-P). Each processor can process its own patterns independently from the other processors.

2.2.1 Training

Once the patterns are loaded, each processor can operate in exactly the same way as a stand-alone processor and can run independently from the other processors, but for one exception: during the training phase. During the training phase, the network is changed. If each processor runs independently from the others, without communication, each processor would end up with a network

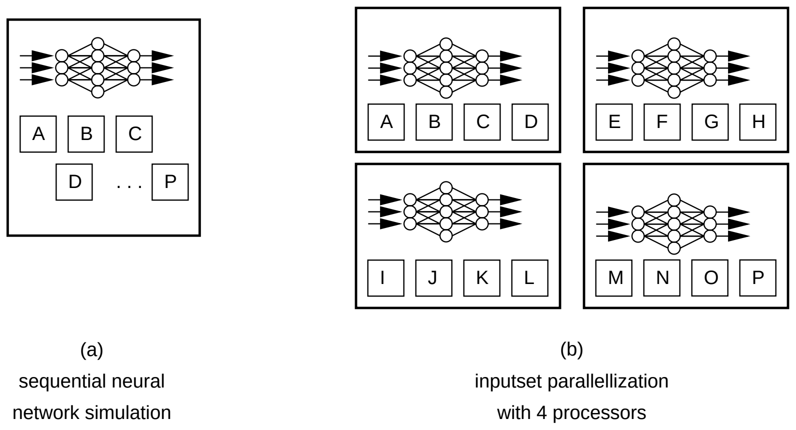


Figure 2.2: Inputset decomposition in a neural network simulation

trained solely for its own subset of the inputset, which will definitely differ from a network trained for the complete inputset.

In figure 2.2b, the top left processor, for example, would only be trained for patterns A to D, not for the other patterns. Therefore, during training, all processors should not only include the changes to the network made locally, but also the changes made by the other processors.

Training per pattern

Remember the training algorithm for training per pattern, given in chapter 1. A first approach to the inputset parallel version of that algorithm is given below.

```
WHILE NOT well trained DO
    FOR EACH pattern DO
        calculate net activation(pattern)
        calculate net error
        calculate network updates
        gather updates from other processors
        update network
    END FOR
END WHILE
```

The only change in the algorithm is the insertion of the line printed in *italic*. Just before the network is updated, the changes derived by the other processors are gathered and added to the changes derived locally. The total change is finally applied to the network. Note that the overall behaviour of the simulation program will differ from the behaviour of the sequential version, as more than one pattern is evaluated in one training step. The result lies somewhere between the original training per pattern and training per epoch.

Unfortunately, the algorithm above is too simple. In general, the amount of patterns will not be the same on all processors, which means that a processor with less patterns will miss the last updates made by other processors, or will not meet the error criterium (well trained) at the same epoch as other processors.

Therefore, before the actual training is started, each processor should know how many training steps each epoch consists of. The number of training steps per epoch is the maximum amount of patterns found on a processor. When a processor has processed its own patterns it should continue gathering network updates from other processors until all other processors are finished with the current epoch.

```
gather max nr of patterns per processor
WHILE NOT well trained DO
    FOR EACH pattern DO
        calculate net activation(pattern)
        calculate net error
        calculate network updates
        gather updates form other processors
        update network
    END FOR
    REPEAT max nr of patterns - own nr of patterns TIMES
        gather updates from other processors
        update network
    END REPEAT
END WHILE
```

Training per epoch

Training per epoch is easier to implement. The network changes are only applied once per epoch, at the end, and it suffices if the changes of the other processors are gathered just before that network change.

```
WHILE NOT well trained DO
    FOR EACH pattern DO
        calculate net activations(pattern)
        calculate net error
        calculate network updates
        accumulate network updates
    END FOR
    gather updates from other processors
    update network
END WHILE
```

2.2.2 Recall and Classify

The algorithms for recall and classify do not need any adaption for inputset parallelization, apart from the fact that the test and classification results are to be gathered by the master processor to make them available to the user.

2.3 Network Decomposition

Network decomposition is the second form of dataset parallelization of neural network simulation programs. Instead of splitting the inputset, the network data is distributed over several processors. The easiest and most flexible way of doing this, is by allocating on each processor a number of neurons from each layer of the complete neural network, plus the belonging weights and other datastructures. In this way, each slave can operate in the same way, but on a different part of the network. Look at figure 2.3 for an example. The network has three layers with three, five and three neurons in each layer, resp. When distributed over a master and two slave processors, each processor gets a more or less equal share of the neurons of each layer and the weights that belong to those neurons.

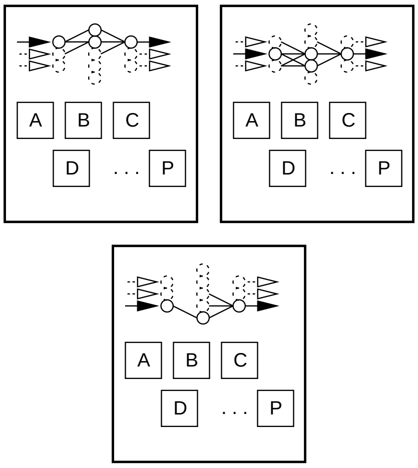


Figure 2.3: Network decomposition with three processors

2.3.1 Training, Recall And Classify

Each processor operates on only a part of the neural network; a part that is exclusively located on that processor. This implies that, in contrast to the inputset decompositioning method, no special measures have to be taken to 'synchronize' the neural network across the processor topology. The training algorithms that are presented in chapter 1 apply unchanged. The parallelism is hidden in the implementation of calculate net activation, calculate net error etc. The same holds for the algorithms for recalling and classifying.

Chapter 3

PREENS And CONVIS

This chapter will introduce PREENS and CONVIS. PREENS, an acronym for *P*arallel *R*esearch and *E*xecution *E*nvironment for *N*eural *S*ystems, is a neural network simulation environment with many capabilities. CONVIS is the front-end of PREENS and gives *con*trol and *vis*ualization of the neural network simulation PREENS is running.

3.1 History of PREENS

PREENS was initiated in January 1990, as a part of the European ESPRIT Parallel Computing Action (project 4106) and the Dutch Stichting Neurale Netwerken (Foundation for Neural Networks). Its main goal is to support researchers in working with neural networks. It does so by providing tools for the construction, visualization, manipulation and simulation of neural networks. PREENS is designed to run on a wide variety of machine architectures, but the main target platform is the Nijmegen SuperCluster transputer system (NSC), which is discussed in chapter 4.

Originally, PREENS should consist of three components, each targeted at a specific part of the design and implementation of a neural network simulation program:

network construction For the construction of a neural network, a network description should be made, written in a neural description language (NDL). To create a network description an NDL syntax directed editor could be used or some kind of graphical editor in which the user can manipulate graphical representations of the building blocks used in NDL.

simulation construction This part should comprise a compiler for NDL and a Process Distributor, which uses a library of network routines and a processor configuration to build a C (or parC) program that is designed to run on the NSC.

execution environment Finally, the compiled C program is run on the NSC or one of the other target platforms PREENS was designed to run on). The simulation program communicates with the execution environment, where the user can view the datastructures in the simulation, start tools and monitor variables and data etc.

The complete original PREENS architecture is depicted in figure 3.1.

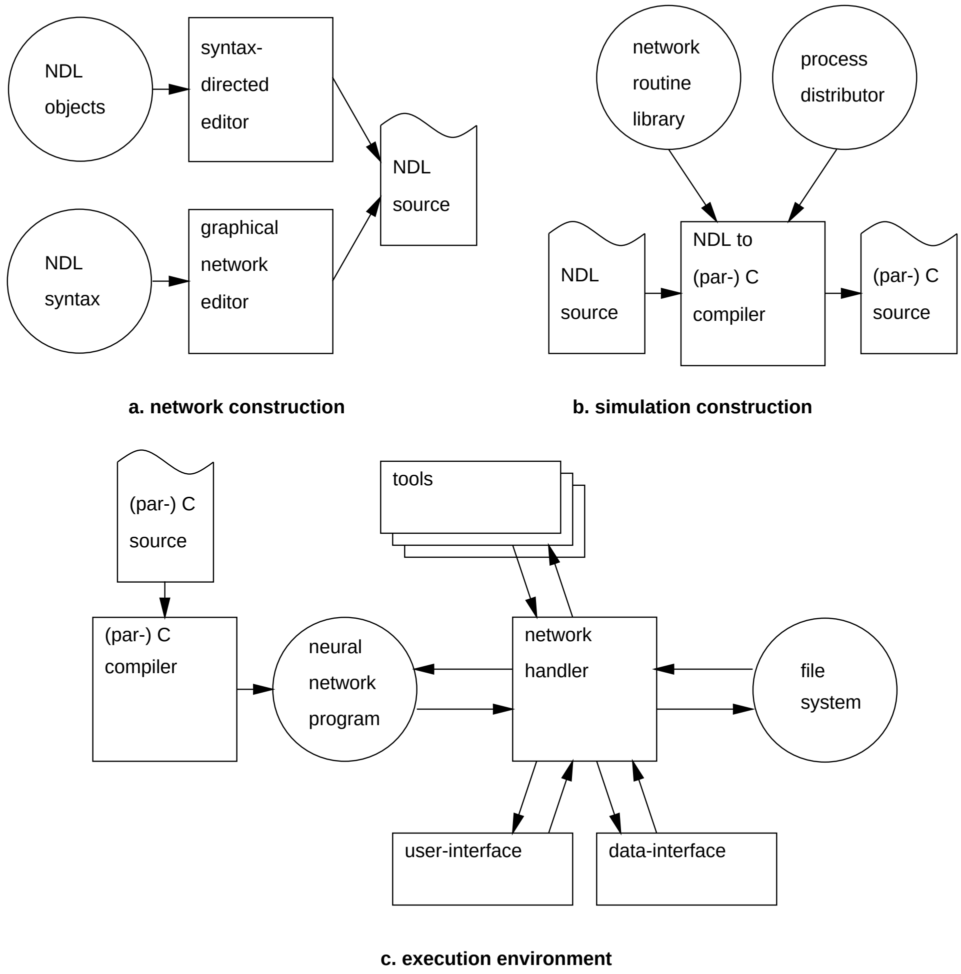


Figure 3.1: The original layout of PREENS

In time, the main focus of attention moved from the first point of the above list, the construction of the network and the design of an appropriate NDL, towards the last point, the execution environment. The absence of an NDL implies that the user can implement his neural network in his own favorite programming dialect and link his code to the interfacing elements for communication with the execution environment [Sch91].

3.2 CONVIS

Experience tells that only about five to ten percent of the code needed for a neural network simulation program is actually code for the simulation. The major part of the code consists of the control and visualization routines and the user interface. Furthermore, the non-simulation part of the code is greatly independent of the neural network simulated, offering the same functionality for most neural network simulations, like loading and saving of patterns.

This observation has lead to a new phase in the development of PREENS. With part c of figure 3.1 in mind, a front-end was developed, that would offer a graphical user interface and an interface to in- and output tools for variables and datastructures. This front-end is called CONVIS. Along with the development of CONVIS a library of auxiliary functions was created that would aid the neural network programmer in fast program development.

CONVIS now serves as the heart of PREENS, and PREENS 'grows' around CONVIS. When running, PREENS now looks like depicted in figure 3.2.

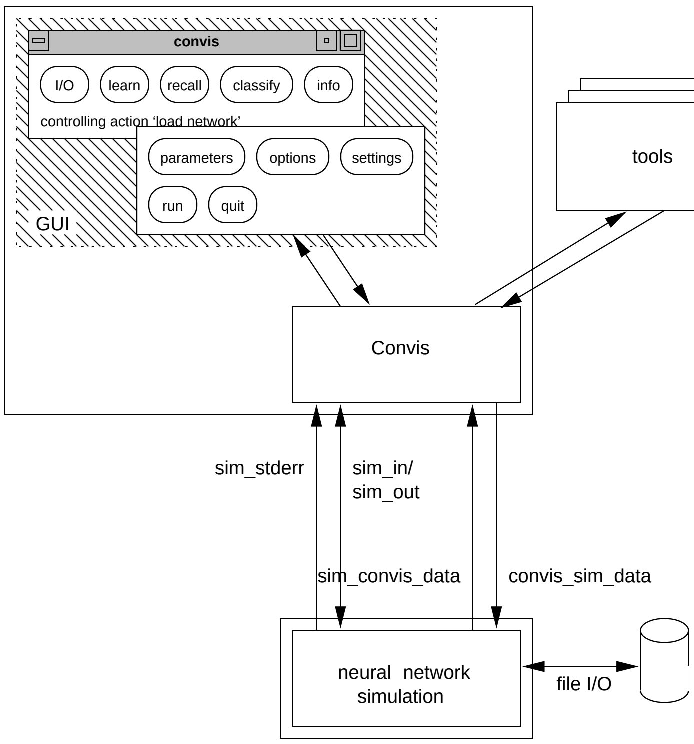


Figure 3.2: CONVIS/PREENS in action

3.2.1 Action Orientation

The basic idea of CONVIS is, that a neural network simulation is formed around a relatively small amount of *actions*. Typical actions are training, building a new network, storing a network to file etc. Each action has associated with it a number of objects:

parameters Scalar values that can be set before the action is started.

variables Scalar values that can change during the execution of the action.

data Non-scalar objects.

options One-of-many selectors and

settings on/off switches that can control the operation of the action.

Consider for example an action 'learn', that trains the network. Parameters of this action could be the maximum number of training epochs, the learning rate or the error criterium. A typical variable for this action is the network error, which must drop below the error criterium to finish the action. Training data are, of course, the training patterns, the weights and neuron activations. An option could indicate whether training per pattern or training per epoch should be performed, while a setting could indicate that the network has to be saved each epoch.

Each object, in turn, has a number of features, like a name, an initial value and boundaries between which the value has to be. Each of these features can be requested or updated by the user of CONVIS. A monitoring tool can for example read the value of the net error during training, or the user can set the name for the network file.

While running, both the simulation program and CONVIS maintain a database in which all these objects are stored, grouped by action. Via the user interface, an action is selected by the user. The objects associated with the action can be set, and the action can be run.

3.2.2 Message Passing

CONVIS and the simulation program remain separate programs, which interact via several communication channels. The advantage of this set-up is, that the simulation program can run on a machine other than the one CONVIS is running on—the machines can even have a different architecture—and makes it easy to connect CONVIS to other simulation programs.

Via the communication channels, implemented as socket streams, messages are exchanged. Messages sent by CONVIS and the simulation program, have a fixed format:

```
STRUCT CMessage =
    INT command
    INT identification
    INT index
    LONG FLOAT value
    INT size
END STRUCT.
```

The first field, command, contains the command to be executed. CONVIS defines a wide range of commands that include commands for requesting or updating data, starting and stopping actions, etc. The identification and index field define which action and which object of the action the command should be executed on. If the command defines that the value of an object should be changed, the new value is contained in the value field. Each message may be followed by an arbitrary amount of unstructured data. The size of this data is stored in the size field.

These messages accomplish a kind of handshake protocol. CONVIS or the simulation program initiates the handshaking by sending a message with a certain command, the other replies by sending a message back, with the identification field set to either an OK or not-OK value, depending on whether the command was successfully executed or not.

3.2.3 Communication channels

The neural network simulation program and CONVIS communicate via six channels:

sim_in, sim_out These two file streams are used to exchange CONVIS messages. The simulation reads a message from sim_in, processes it and sends a reply via sim_out.

sim_stderr This is the CONVIS error channel. All messages sent through this file stream will appear in a dialog box on the user's screen, called 'Error Messages'. Note that this is not the same as the stderr stream—messages to stderr will appear on the shell the simulation was started on, unless it was redirected.

convis_sim_data While an action is running, variables and data may be changed by an input generating tool. The values generated are transmitted by CONVIS and received by the simulation program via this stream.

sim_convis_data This is the stream for exchanging data and variables the other way around. Data and variables that have to be monitored while an action is running are sent via this channel.

sim_interrupt The user may interrupt the running action before it is properly finished. When an action is started, a daemon process is spawned that watches this channel and raises a flag when a message arrives via sim_interrupt. The running action notices the raised flag and will process the message received.

3.3 The Current Interface

In figure 3.3 the main components of a neural network simulation program built with CONVIS are shown. Only two elements of it are specified by the neural network simulation programmer, sim and sim_functions. Sim contains the main program, which only calls some functions of parse_actions and interface. The right hand side of the area beneath sim contains the routines for reading and processing the program description. An example of a CONVIS program description file is given in appendix A.

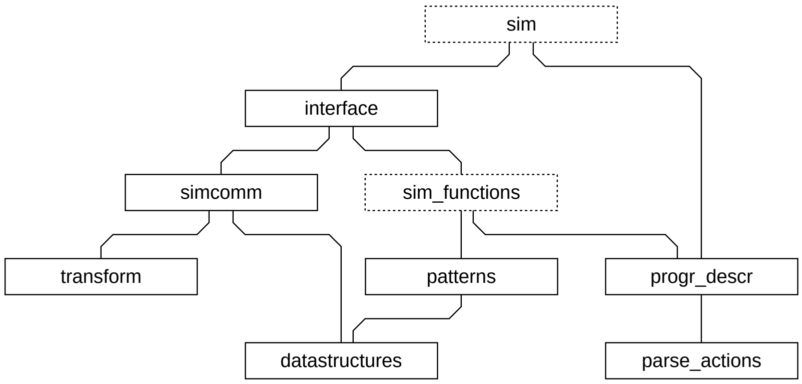


Figure 3.3: Build-up of a CONVIS neural network simulation program

The left hand side is the part which has to be adapted for use with a parallel neural network simulation program. It consists of several parts of which four are of interest with respect to the design of the parallel version of the interface:

interface contains the routine that handles incoming messages from CONVIS. Each incoming message is checked on its contents (values of the identification and command fields (see page 22) and appropriate action is taken.

simcomm defines the communication routines between the simulation program and CONVIS. The routines that set up the communication channels are also in this module.

transform defines routines that transform CONVIS action objects (parameters, variables, etc) into unstructured buffers and vice versa. These buffers can be sent as an attachment to the messages sent to CONVIS and can receive attachments that follow the messages received.

datastructures contains routines for handling datastructures. Data can be allocated, buffered, deallocated and debuffered with routines specified in this module.

patterns defines routines for loading and saving patterns for the simulation program in several formats. By loading patterns in one format and saving them in another, conversion between the formats is possible.

The remaining two modules, sim and sim_functions are defined by the programmer. Sim contains the main routine of the neural network simulation program and is almost the same for each simulation program. Sim_functions contains all the action functions that are necessary for the neural network simulation. This part is completely implemented by the programmer.

Chapter 4

Parallel Computing On The NSC

The Nijmegen SuperCluster (NSC) transputer system is a parallel computer, designed for high-performance computing. In this chapter, a brief description of the inside of the NSC will be given and an explanation will be given how to set up the NSC to run a certain parallel application.

4.1 The Hardware And The Operating System

The NSC is a transputer system built and installed by Parsytec consisting of sixty-four Inmos T800 transputers. A transputer is a microprocessor which is designed to operate in a multi-processor environment. It has a number of bi-directional high speed communication links with which it can communicate with other transputers. The T800s used in the NSC have four links. In the NSC, the transputer links are connected to a set of network configuration units (NCUs). By means of these NCUs the transputers in the NSC can be connected to each other into any desired configuration.

The NSC is not part of the KUN campus ethernet LAN. Instead, it is accessed via a dedicated host computer, which is part of the ethernet LAN. The host computer is a Sun4 computer, running SunOS (UNIX) and equipped with an extension board. This extension board carries eight transputers which connect, by means of an additional NCU on the extension board, to the NCU's inside the NSC. Via these eight transputers, up to eight users can simultaneously make use of the NSC.

The NSC runs a particular operating system, Helios, which is very similar to UNIX, but which has special provisions for multi-processor environments. Helios implements a major part of the POSIX standard for UNIX, but not all of it. The differences are mostly a consequence of the multi-processing capabilities incorporated in Helios.

4.2 Building A Topology

When a user logs in on the NSC (or actually on the transputer system on the extension board inside the SUN host), the first thing he has to do is define the transputer configuration or topology he wants to use. A means is provided to define such a configuration and that means is the *resource map*. The resource map can define what processors are used, how these processors are linked together, what resources they use and how they communicate. An example resource map is given in figure 4.1. It defines a ring topology with four processors.

Each terminal part defines one processor in the topology. The part above the comment '# user part' is the same for all resource maps; it defines the part of the topology that interfaces the user part with the 'outer world'. The SUN host acts as the I/O processor (terminal /IO) in the topology, terminal /00 is the transputer on the extension board in the host, that is appointed to the user of the NSC.

Per processor, first the links are defined. Processor /02 for example, connects to processor /03 with link 1, to processor /01 with link 3, and has no connections to link 0 and 3. Next, the processor type is given; all processors that are free for the user have type HELIOS, the processor that performs the in- and output, has type IO and the connection between the I/O-processor and the rest is laid by a processor of type SYSTEM. After that, the processor type and the minimal memory requirements are specified.

This resource map can be compiled with rmgen, which produces a binary form of the map. This map is read at login time, when the user of the NSC has to specify the topology he will be using. The NCUs in the NSC will allocate the requested processors and build the topology by connecting the right links.

During log in, the user can request a 'single processor session' instead of a multi-processor topology. In that case the topology will only consist of the I/O processor, ie. the Sun4 host, and one transputer on the extension board. The user will thus not use any of the sixty-four transputers in the NSC, and will not be able to run parallel software. The /00 processor will then run the user's commands.

4.3 Running A Parallel Application

Once the user is logged in, a shell is started on the root processor—the processor directly connected to the system processor—and the user can start his application. The Helios shell is equipped with a dedicated script language, CDL [Vee90], which stands for *C*omponent *D*escription *L*anguage. CDL is based on communicating sequential processes: a parallel program is constructed by coupling sequential processes via communication channels. It is obvious that this model applies directly to the transputer philosophy.

```
# resource map for a 4 point ring

subnet /Cluster {
    CONTROL Rst_Anl [/Cluster/00];

    # system part

    terminal /IO {
        ~00  ; IO;
    }
    terminal /00 {
        ~IO , , ~01 , ; SYSTEM;
        Mnode Rst_Anl [pa_ra.d];
        ptype T800;
    }

    # user part

    terminal /01 {
        ~00 , ~02 , , ~04 ; HELIOS;
        ptype T800;
        memory #400000;
    }
    terminal /02 {
        , ~03 , , ~01 ; HELIOS;
        ptype T800;
        memory #400000;
    }
    terminal /03 {
        , ~04 , , ~02 ; HELIOS;
        ptype T800;
        memory #400000;
    }
    terminal /04 {
        , ~01 , , ~03 ; HELIOS;
        ptype T800;
        memory #400000;
    }
}
```

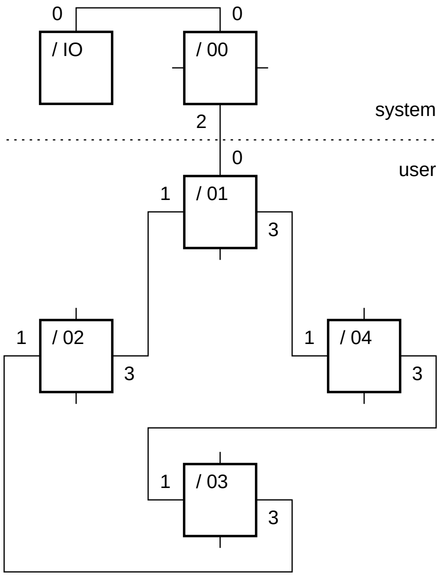


Figure 4.1: A 4 transputer ring topology.

4.3.1 Parallel Constructors

A CDL script defines a parallel program, in Helios terms called a ‘taskforce’, as a combination of sequential processes that are connected via special parallel constructors. There are four of these constructors, each defining a particular way of communication between its two operands. (See figure 4.2)

A <> B This defines a bidirectional pipeline between A and B.

A | B This defines defines a uni-directional communication line, a pipe, from A to B, similar to the pipe constructor used in UNIX shells.

A ||| B The *interleave* constructor defines a load balancing process between A and B. This operator is mainly useful in combination with replicators.

A ^^ B This defines two parallel processes, but defines no communication channels between A and B.

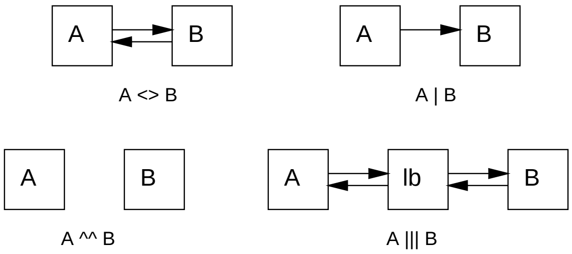


Figure 4.2: Parallel constructors

It is possible to construct more complex taskforces by combining the constructors, very similar to making expressions with the normal arithmetic symbols. The parallel constructors have an order of precedence, in which they were listed above, with <> having the highest priority and ^^ the lowest.

The replicator mentioned, is a means to help specify taskforces consisting of replicating elements. With the help of replicators, the specification of, for example, a master process A and a farm of seven slave processes B is denoted as A [7]||| B.

4.3.2 CDL scripts

The most important feature of CDL scripts lays in the possibility to define subtasks, called components. Each component describes exactly one process and can contain annotations on the communication streams used, the node in the transputer topology the component must be placed on and the code that should be run for the component. An example CDL script that runs a master process and three slave processes on the topology defined in figure 4.1 can be found in figure 4.3.

```
# master and 3 slaves in a ring

# the components

component master {
    streams , , , ,
        <| ms1, >| sm3,
        <| ms3, >| sm1,
        <| ms2, >| sm2;
    puid /Cluster/01;
    code master;
}
component slave_1 {
    streams <| s12, >| ms1, , ,
        <| sm1, >| s21;
    puid /Cluster/02;
    code slave;
}
component slave_2 {
    streams <| s23, >| s12, , ,
        <| s21, >| s32,
        <| ms2, >| sm2;
    puid /Cluster/03;
    code slave;
}
component slave_3 {
    streams <| sm3, >| s23, , ,
        <| s32, >| ms3;
    puid /Cluster/04;
    code slave;
}

# and the taskforce definition

master ^^ slave_1
       ^^ slave_2
       ^^ slave_3
```

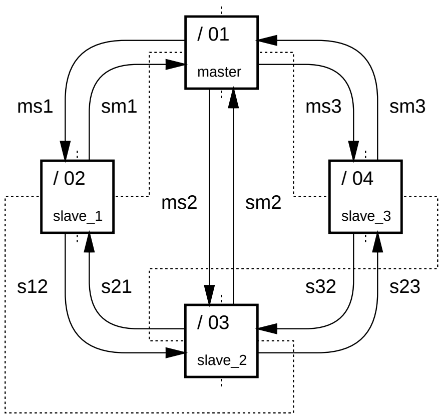


Figure 4.3: A sample CDL script for the ring topology

Each process, or component, in the parallel program is defined by the component keyword. For every component, the processor it will run on is defined by the puid keyword. Defining the puid is not mandatory, but it forces the parallel program in a well known topological mapping. When the puid is not specified, the taskforce manager will select a suitable processor. The process code that must be run for a certain component is specified by the code keyword. In figure 4.3, the master component, placed on processor /Cluster/01, runs the master code, all slaves run the slave code.

Communication channels are defined by the streams keyword. The streams defined here correspond directly to the file descriptors used in the program. The first stream specified in the streams section corresponds with file descriptor 0, usually the standard output, the second stream corresponds with file descriptor 1, the standard input channel, etc. Streams that are not specified are free.

In the example of figure 4.3, each slave uses its standard output (0) and standard input (1) for the counter clockwise pipeline, keeps the standard error (2) and standard auxiliary (3) free and uses stream 5 and 6 for the communication in clockwise direction. The master keeps his first four streams free. These will be used as the standard streams of the complete taskforce.

Note that direct bidirectional communication between the master and slave_2 is also possible despite the fact that the processors they run on are not directly connected. The communication streams will, physically, be performed via one of the other two processors, but this is completely transparent to the CDL taskforce and the processes that run within the taskforce. The noticeable difference between a direct and an indirect communication link will only lie in performance. Communication via a direct link will, of course, be faster than communication via an indirect link.

The last lines in the CDL script define the taskforce to run. The master and slave components are run in parallel, as defined by the ^^ operator. No communication has to be specified in the taskforce definition as all communication channels are laid down in the component definitions.

Chapter 5

The Parallel Interface For Convis

In this chapter, the design of the parallel interface will be given. First, the modules added to the interface as given in chapter 3 will be described. Then, the implementation on the NSC will be given and the adaptions made to other modules will be described.

5.1 The Parallel Interface

The extensions made to the existing interface are depicted in figure 5.1. Three modules are added: transputer, topology and distribution.

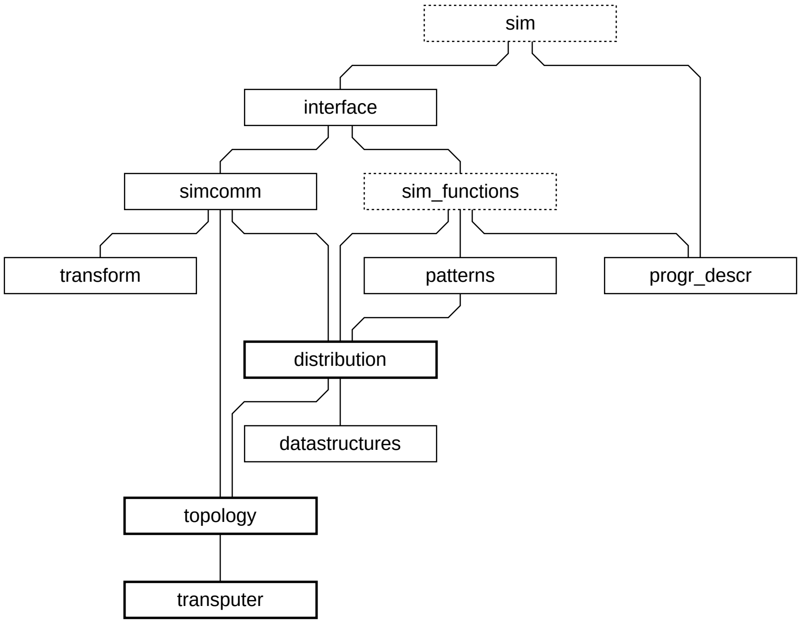


Figure 5.1: Modularization of the parallel interface

The module transputer contains the definitions and routines needed for inter-transputer communication. The routines here form the communication primitives for the other modules.

Topology is the module which defines the topology dependent communication routines. For each topology a suitable, and probably optimized, module can be defined. This module is used by simcomm for the broadcasting of messages from CONVIS and by distribution for the communication of data.

The third module added, distribution, defines routines for distributing and gathering data. The routines in this module can be optimized for a certain topology and will therefor be strongly related to the topology module.

5.2 Description Of The Newly Added Modules

5.2.1 Transputer

The module transputer contains the routines for communication within the processor topology. Its design is focused on implementation on the NSC, but it should be easy to port to other parallel computing platforms.

Link names and connection routines

A transputer has four links for communication. By convention, each link is given a direction/orientation as name. The routines (dis)associate a link name with two communication channels.

```
TOP,UP      NORTH
LEFT,       WEST
DOWN,       SOUTH
RIGHT,      EAST

ConnectLink(link, fd_in, fd_out)
DisconnectLink(link)
```

TOP, LEFT, DOWN and RIGHT are the names to be used for the link parameters. UP, NORTH, WEST, SOUTH and EAST are aliases for the first four names.

ConnectLink connects file descriptor fd_in and fd_out to a communication link; fd_in will be used for reading from link link, fd_out for writing. DisconnectLink closes the file descriptors associated with link link.

Communication primitives

Three routines form the very core of the communication library in the parallel interface. A neural network simulation programmer will seldom use these as more elegant routines are available.

FullRead(link, buffer, size)
FullWrite(link, buffer, size)
Transpute(from, to, size)

FullRead and FullWrite read and write size bytes from and into buffer, respectively. Transpute reads size bytes from link from and writes them to link to. It uses an internal buffer, so there's no need to specify a buffer as parameter.

Typed communication routines

A couple of higher level communication routines is defined for each of the six primitive types defined by CONVIS. The routines for the Byte type are given here. Similar routines exist for Short, Int, Long, Float and Double:

ReceiveByte(link, buffer)
SendByte(link, buffer)
TransputeByte(from, to)
ReceiveNBytes(link, buffer, size)
SendNBytes(link, buffer, size)
TransputeNBytes(from, to, size)

These routines are nothing more than wrappers around the primitive communication routines, but they facilitate programming a lot.

5.2.2 Topology

With the routines in transputer, the processors in the parallel neural network simulation program can communicate with each other. No assumptions are made on how the processors are connected. In the module topology, the routines that define or depend on the topology are defined.

Variables to examine the topology

A number of variables is specified which can be used to get information on the topology.

is_master, slave_id
has_top, has_left, has_down, has_right
nodes_top, nodes_left, nodes_down, nodes_right
nnodes

The variables is_master and slave_id can be used to identify the processor the program is running on. The master processor will have -1 as slave_id and is_master set to a non-zero value; the slaves will be numbered 0 and up. The exact numbering of the slaves depends on the topology used.

The variables has_top, has_left, has_down and has_right indicate whether the processor has a connection with another processor in the indicated direction. The position of the processor in the topology can be determined using the variables nodes_top, nodes_left, nodes_down and nodes_right. The exact meaning of these variables may differ slightly, depending on the topology used. The last variable, nnodes, contains the number of processors in the topology.

The values of all these variables are set during the initialization of the neural network simulation program, when SetupTopology is called.

Routines to set up the topology

The routines are defined to set up or close down a transputer topology.

```
SetupTopology(argc, argv)
CloseTopology()
```

SetupTopology checks the processor topology. If the expected topology and the determined topology are not consistent, an error is given and the simulation program may be ended. This routine must be called at the very start of the program. If the topology is found to be ok, the variables mentioned above are set to their appropriate values. The arguments of the routines SetupTopology are the commandline argument counter (argc) and the command line arguments (argv), as the command line may contain information on the topology setup.

CloseTopology closes all communication channels. It should be called when the the simulation program is exited.

Topology dependent communication routines

These routines form the basis of topology wide communication. More routines may be added, but these are necessary for the interface to run. The routines for the Byte type are given, but the same functions exist for the other types defined by CONVIS.

```
GatherByte(buffer)
BroadcastByte(buffer)
GatherNBytes(buffer, size)
BroadcastNBytes(buffer, size)
```

The broadcast routines will take the buffer from the master processor and copy this to all slave processors. The gather routines are a sort of complement to this: the values in the buffers of the slaves are summed and the master will hold the result. Formally specified, the result of the gather routine for master processor is:

$$\mathsf{buffer}_m[i] = \sum_{n=0}^{\mathsf{nnodes}-1} \mathsf{buffer}_n[i]$$

where buffer_n is the buffer on slave n, and buffer_m is the master's buffer. The result for the slaves is undefined, so often a gather operation will be followed by a broadcast to update the buffer on all slave processors.

5.2.3 Data_distribution

The module data_distribution contains the routines for distributing the data. These routines may depend on the topology used.

MakeDataDistribution(dat)
AllocateDistributedData(dat)

SendDistributedDataPart(message)
ReceiveDistributedDataPart(message)

The first two routines are responsible for the distribution of data structures. The dat parameter is a pointer to a CONVIS data description (CData). MakeDataDistribution defines how the data should be distributed over the available processors. AllocateDistributedData allocates the data per processor. Distributed data must be received from or sent to CONVIS using the Receive- and SendDistributedDataPart routines.

This is a minimal definition of the data_distribution module. More functions can be added to facilitate further neural network simulation development.

5.3 Implementation Of The New Modules

5.3.1 Implementation Of Transputer

The implementation of the transputer module on the NSC is very straightforward. The file descriptors used in ConnectLink are already opened by the CDL script when the program starts and will be closed automatically when the program terminates. Consequently, ConnectLink only checks whether the given filedescriptors are really opened, and DisconnectLink just marks the links as disconnected, but leaves the file descriptors open.

5.3.2 A Ternary Tree Topology

As the transputers in the NSC have four communication links, it is possible to build a ternary tree topology. The implementation uses a node numbering scheme that ensures that for any given number of nodes, a well-balanced tree will be built. For a node n its sons are defined by the following equation:

$$\mathrm{son}[l] = n + l.3^{\mathrm{depth}(n)}, \quad l \in \{1, 3\}$$

where *depth(n)* is the depth of the node in the tree. The depth of a node can be calculated with the equation:

$$\begin{aligned}\mathrm{depth}(n) &= d \in \mathbf{N}\,[\mathrm{first}(d) \leq n < \mathrm{first}(d+1)]\\ \mathrm{first}(d) &= \begin{cases} -1 & \text{, if } d = 0\\ \sum_{i=0}^{d-1} 3^i - 1 & \text{, if } d \neq 0\end{cases}\end{aligned}$$

With the above equations, also the father node of a given node n can be derived:

$$\mathrm{father} = n - \left(\left\lfloor 3.\frac{n - \mathrm{first}\big(\mathrm{depth}(n)\big)}{3^{\mathrm{depth}(n)}}\right\rfloor + 1\right).3^{\mathrm{depth}(n-1)}$$

Figure 5.2 shows an example of a tree topology built this way.

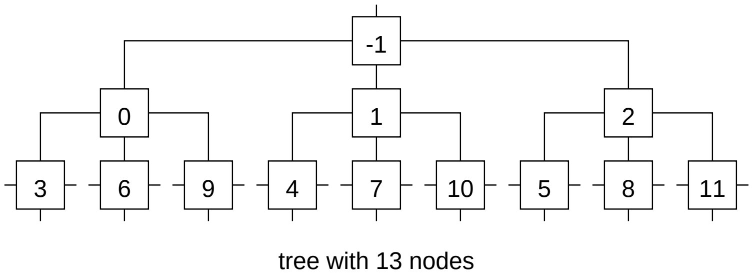


Figure 5.2: Example of a tree topology

Each node uses its top link to communicate with it's father, except for the master, which is the root node and consequently has no father. So, has_top will be true for all slave nodes and false for the master. The value of nodes_top will be set to the *depth* of the node in the tree, while nodes_left, nodes_down and nodes_right will indicate the size of the subtrees in the corresponding directions.

SetupTopology will first verify the physical topology as defined by the resource map (see 4) using the above expressions and the library `RmLib` [Dav91] which is provided by the Helios C compiler especially for purposes like this. If the topology meets the expectations, the communication streams set up by the CDL script will be tested. If any of this fails, the simulation will be ended.

The Broadcast routines receive data from the father node and pass it on to the son nodes. The Gather routines receive data from the sons, perform a elementwise summation on the data buffer and send it to the father node.

5.3.3 Geometrical Data Distribution In A Tree

For the NSC, an implementation of the module data_distribution was built using geometrical data distribution in a tree topology. A geometrical distribution is a distribution in which the distributed parts of the data structure have the same structure (geometry) as the original one.

Figure 5.3 shows a 5×5 matrix, a geometrical and a non-geometrical distribution of it over 3 processors.

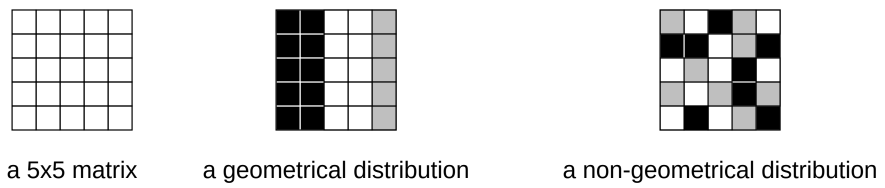


Figure 5.3: A 5×5 matrix, distributed over 3 processors

Distribution of PREENS data

The main datastructure used in PREENS is the multi-dimensional array. A multi-dimensional array $A(d_0, d_1, \ldots, d_m)$ can be seen as a vector $A'(d_0)$ with structured elements $A''(d_1, \ldots, d_m)$. A geometrical distribution of A' will also be a geometrical distribution of A. This means it suffices to specify the distribution for a vector, a one dimensional array. The distribution of multi dimensional arrays is completely analogue.

To get an even workload over the available processors, each processor should get an (almost) even share of the data distributed. If m is the size of the vector to distribute and P is the number of processors in the topology, the number of elements that processor p, numbered -1 to $P-2$, will get ($\#p$) equals:

$$\#p = \begin{array}{ll} \lfloor \frac{m}{P} \rfloor + 1 & , \text{if } p+1 < m \bmod P \\ \lfloor \frac{m}{P} \rfloor & , \text{if } p+1 \geq m \bmod P \end{array}$$

The elements of A allocated on processor p will now range from $lowb_p$ upto $uppb_p$, where $lowb_p$ and $uppb_p$ are defined as:

$$\begin{array}{rcl} \mathrm{lowb}_p & = & \sum_{i=-1}^{i<p} \#i \\ \mathrm{uppb}_p & = & \mathrm{lowb}_p + \#p \end{array}$$

The distribution and gathering routines

The data distribution and gathering algorithms for the ternary tree topology are not very complicated. For the gathering of distributed data, each processor first sends its own data to the top link. Then, data from the left, down and right links is transputed in turn.

```
GatherDistributedData:
    Send own data to top
    DO
        Transpute data from left to top
        Transpute data from down to top
        Transpute data from right to top
    UNTIL no more data to transpute
```

```
DistributeData:
    Receive my own data from top
    DO
        Transpute data from top to left
        Transpute data from top to down
        Transpute data from top to right
    UNTIL no more data to transpute
```

The master processor will of course not read its data from the top link, but will get it from CONVIS, an input tool or a file. The gathering routine is completely analogue.

Implementation of tree_geom

The geometrical data distribution method was implemented in a module consisting of a slave and a master part. The two parts differ mainly in the way data is sent to or received from the top link. The master will not use its top link, but will use a file or communication link with CONVIS to send and receive the data.

The module implements the following functions:

```
P_MakeDataDistribution(cdata)
P_AllocateDistributedData(cdata)

P_SendDataPart(fd, cmessage)
P_ReceiveDataPart(fd, cmessage)

P_SaveDistributedData(file, cdata)
P_LoadDistributedData(file, cdata)
```

P_MakeDataDistribution generates a distribution for the data described by cdata. The global data dimensions in the description of cdata must be set, P_MakeDataDistribution will generate the data distribution from this information. An descriptor for the local (per processor) data dimensions will be set up.

P_AllocateDistributedData will allocate the data described by cdata. If necessary, P_MakeDataDistribution will be called to generate the data distribution.

P_SendDataPart and P_ReceiveDataPart will send and receive the data pointed to in cmessage using file stream fd. These routines are not intended for use by the neural network simulation programmer, but are used in the interface routines to communicate data to and from CONVIS.

Distributed data can be saved to and read from a file using the routines P_SaveDistributedData and P_LoadDistributedData. They use normal ASCII files in standard PREENS format.

5.4 Adaptions Made To Other Modules

All other modules needed some modification in order to work as a parallel interface. Some of them were completely redesigned and implemented, others needed only slight adaption.

5.4.1 Progr_descr

The sequential interface had two modules that took care of the program description: parse_actions and program_descr. Parse_actions contained the routines for the reading and interpreting of the program description file. These routines were used by program_description to build the action data structure.

In the parallel interface, these two modules have been put together into one new program_description module. Some of the routines in the sequential interface module became obsolete and were removed, some others were transfered to the module datastructures (See below).

The new program_descr module contains the following functions:

```
FillInSimActions(argc, argv, cfunction_list)
ClearSimActions()
FillInSimInfo(sim_name, version, author, company)
```

FillInSimActions will take the fourth command line parameter of the master process and interpret it as the program description filename, if possible. When this file can not be found, or there are less than four command line parameters, the program description filename will be created from the master processes filename. If that fails, too, the program will be ended.

From the program description file, the program description is read and the actions functions in the cfunction_list will be attached to the described actions and the action data structure is set up. FillInSimActions is usually called once at the very start of the program and put in the main routine of sim. ClearSimActions clears the action data structure.

FillInSimInfo can be used to set some information on the simulation program. The name of the simulation, version number, author of the simulation program, and the companies or institutions name can be filled in and will appear when the user of CONVIS selects the *info* button.

5.4.2 Datastructures

The module datastructures was adapted to be able to use it in combination with the data distribution module. One function was added to the existing routines, to be able to set the data format for a given data structure. Two functions that were originally in program_descr were moved to this module, because they fitted in better. Routines now in datastructures are:

SetDataFormat(id, index, format)
DataSize(cdata_description)
ElementSize(convis_type)
AllocateData(cdata)
AllocateAllData(action)
FreeData(cdata)
FreeAllData(action)
LinkData(src_id, src_index, dst_id, dst_index)
AssignData(ptr, id, index, dim, d0, d1, d2, d3, length, flat)

SetDataFormat is the new function. With this function the programmer can select which data structures of the simulation program, pointed to by id and index, have to be distributed by setting the correct value of format.

DataSize and ElementSize return the sizes of the given data structure resp. the type of the elements in the data structure.

AllocateData allocates the datastructure of its parameter. The dimensions of the data and the data format must be set previously, using the SetDataFormat and AssignData functions. AllocateAllData allocates all data structures belonging to action action, by repetitively calling AllocateData. FreeData and FreeAllData are the complements of the two allocation functions.

LinkData associates two data structures with each other. The data pointed to by src_id and src_index should already be allocated. The data dimensions are copied from this data structure to the data structure pointed to by dst_id and dst_index.

The function Assign_Data sets up the dimensions for the data structure pointed to by id and index. Dim is the number of dimensions of the data. Its value must lie between 1 and 4 as PREENS doesn't support data with more than 4 dimensions. The parameters d0 to d3 and length set the depth of each dimension. The last parameter indicates how the data should be allocated.

5.4.3 Patterns

The module patterns was implemented twice. One version for the input set parallelization was made, and one version for network decomposition. (see section 6.3). The former version uses routines from distribution to allocate and distribute the patterns loaded. The latter just loads the patterns from file to the master processor, which in turn broadcasts them to the slaves.

No functions were added, nor was their external functionality altered:

CreatePatternHeader(header, format, n_patterns, n_inputs, n_targets,
    on_value, off_value, in_min, in_max, tar_min, tar_max)
LoadCheckHeader(file, req_header, file_header)
AllocatePatterns(header, inputs, targets)
FreePatterns(header, inputs, targets)
LoadPatterns(file, header, inputs, targets)

CreatePatternHeader is used to create a data structure (header) which keeps the information for the loading of the patterns. The format parameter indicates the types of the input and target patterns. The number of patterns can be set in n_patterns. The file will not be loaded when the number of patterns in it differs from n_patterns. To load a file with any number of patters, n_patterns must be set to 0.

The sizes of the in- and output patterns must be set in n_inputs and n_targets. For files using a binary pattern representation, the values on_value and off_value will be filled in for an active and inactive state resp. The last four parameters set boundaries between which the values for the patterns in the file must lie—values across those boundaries will be set to the indicated minimum or maximum.

LoadCheckHeader can be used to test a file's contents. Req_header must be filled using CreatePatternHeader and on success, file_header will contain the information about the pattern file file. On failure, either the file could not be recognized as patterns file, or the requirements set in req_header were not met.

The last three functions speak for themselves: AllocatePatterns allocates space for the patterns, using the info in its header parameter, FreePatterns frees the space occupied by the patterns. LoadPatterns reads the patterns from file file and sets up the pattern data structure in inputs and targets.

LoadPatterns does all the necessary allocating and checking, so normally it suffices to set up a pattern header with CreatePatternHeader and consecutively call LoadPatterns.

5.4.4 Transform, simcomm and interface

Although the three modules transform, simcomm and interface were radically redesigned and re-implemented, the external functionality of these modules remained the same as in the sequential interface. These modules will not be discussed here further as neither the user, nor the neural network simulation programmer needs to use functions of these modules.

Chapter 6

CASE: Counterpropagation

As a test case for the parallel interface for CONVIS, a simulation program has been implemented that models the counterpropagation network. This chapter contains a description of the counterpropagation network, the specification of the simulation program and the implementation aspects encountered.

6.1 The Counterpropagation Network

The counterpropagation network was invented by Robert Hecht-Nielsen [Hec90] [Was89, chapter 4] and comprises a mixture between two already known neural network paradigmas. Basically, the counterpropagation network consists of three layers. The first layer is, of course, the input layer. The second layer is a Kohonen self-organizing map (SOM). The third layer, which is the output layer, is called the Grossberg layer. It maps the output of the Kohonen SOM to the desired output patterns.

The counterpropagation network is not as flexible as the backpropagation network, but can be trained much faster. When trained, the Kohonen SOM functions as a classifier, where each Kohonen neuron represents one class. When an input pattern is presented to the network, only the Kohonen neuron with the maximum net input will become active. The Grossberg layer then 'decodes' the class derived to the output corresponding to that class.

6.1.1 Normal Operation

In normal operation, the net inputs are calculated in the most traditional way, as weighted sums. The Kohonen layer operates in a winner-takes-all fashion: only the neuron with the highest net input outputs a logical one, all others produce zero.

Input vectors

Although it is not mandatory, normalizing the input vectors can have a positive effect on the performance of the counterpropagation network. For each input vector $X = (x_0, \ldots, x_{n-1})$, the normalized input vector $X' = (x'_0, \ldots, x'_{n-1})$ is calculated as:

$$x'_i = \frac{x_i}{\sqrt{\sum_{k=0}^{n-1} x_k^2}}$$

Kohonen layer

The net input of the Kohonen layer is, in matrix notation:

$$L = X \cdot W$$

where L is the net input vector of the Kohonen layer, X is the input vector and W the weight matrix between the input and the Kohonen layer.

The net activation can be calculated as follows:

$$K_i = \begin{cases} 1 & \text{if } i = \max(L) \\ 0 & \text{otherwise} \end{cases}$$

where K is the net activation vector of the Kohonen layer and max is a function that outputs the index of the maximum element in its vector argument.

$$\begin{aligned} \max &: \mathrm{R}^n \mapsto \underline{n} \\ \max(X) &= i \mid \forall j \in \underline{n}\, [X_i >= X_j] \end{aligned}$$

Grossberg layer

The activation of the neurons in the Grossberg layer is the same as their respective net inputs:

$$Y = K \cdot V$$

where Y is the Grossberg layer net input and activation vector, K is the Kohonen activation vector and V is the weight matrix between the Kohonen and the Grossberg layer.

6.1.2 Training

Training of the counterpropagation network can be split in two parts: training the Kohonen SOM and training of the Grossberg layer. The two training steps are performed simultaneously.

Kohonen layer

To train the Kohonen layer, an input vector X is presented to the input layer and the net activation is calculated. The weights of the winning neuron in the Kohonen layer are then adjusted using the following equation:

$$w'_i = w_i + \alpha \cdot (X_i - w_i)$$

where X_i is the i-th component of the input vector X, w_i is the old value of the weight connecting the winning neuron to X_i, w'_i the new value and α a training coefficient. The value of α should start at approximately 0.7 and may be gradually decreased during the training process.

Grossberg layer

The Grossberg layer is trained supervised: for each input vector X in the set of training patterns a corresponding target output vector Y exists. The Grossberg layer is trained to output Y when X is presented to the counterpropagation network.

Training is performed by presenting an input vector X to the network and calculating the net activation. The weights in the weight matrix V are then adjusted as follows:

$$V'_{ij} = V_{ij} + \beta \cdot (Y_j - V_{ij}) \cdot K_i$$

where K is the activation vector of the Kohonen layer and V_{ij} the the weight connecting Kohonen neuron i to Grossberg neuron j. Note that for only one Kohonen neuron K_i is 1 and that only the weights connecting to this winning neuron are allowed to learn.

6.1.3 Variations

There are some variations on the standard operation of the counterpropagation network as presented above. The application of these variations may improve the performance of the network.

Accretive and interpolative mode

The operation of the counterpropagation network as presented in section 6.1.1, where only one neuron in the Kohonen layer is activated is called the *accretive* mode. The capabilities of the counterpropagation network in accretive mode are limited as the output of the network is a function of only one neuron in the Kohonen layer.

In *interpolative* mode, a group of Kohonens having the highest output are activated. The output of the Kohonen layer is then characterized by:

$$k_i = \begin{cases} \frac{L_i}{|L|} & \text{if } i \in \max'(L, m) \\ 0 & \text{otherwise} \end{cases}$$

where m is the number of Kohonen neurons that is allowed to win and $|L|$ is the length of the net input vector of the Kohonen layer. In this way the Kohonen activation vector is of unit length. The function $\max'$ is a variation of the max function shown earlier; instead of the index of the Kohonen neuron with the maximum input, it delivers a set with indices of m neurons having a maximum input.

$$\begin{array}{rcll} \max' & : & (\mathrm{R}^n, \underline{n}) \mapsto \{\underline{n}\} & \\ \max'(X, m) & = & A \;\; | & \forall i \in A \; [\forall j \in \underline{n} \backslash A \; [X_i >= X_j]], \\ & & & |A| = m, \\ & & & \forall x, y \in \underline{n}, \; [(X_x = X_y \wedge X_x \in A \wedge X_y \notin A) \rightarrow x < y] \end{array}$$

The last constraint ensures that for every X and m, $\max'(X, m)$ is unique.

The convex combination method

The goal of training the Kohonen layer is to make the weight vectors resemble the normalized input vectors. This causes the Kohonen neuron whose weight vector resembles the input vector the most to have the maximum net input and thus become activated.

The weights of the Kohonen neurons are usually initialized at random. This performs well if the input patterns are more or less equally spread through the input space. However, if several input patterns are very similar, but must be classified in separate classes, it may be difficult to get the network trained that way. The *convex combination method* may solve this problem.

Training of the Grossberg layer is the same as explained above, but that of the Kohonen SOM differs. At initialization, all Kohonen weight vectors are set to the same value $1/\sqrt{n}$, where n is the number of input neurons. This makes all weight vectors of unit length and coincident. During the training, the input vectors are preprocessed as follows:

$$x_i' = \gamma \cdot x_i + (1 - \gamma) \cdot \frac{1}{\sqrt{n}}$$

At the start, γ is chosen very small, causing all input vectors to have a length near $1/\sqrt{n}$ and almost the same as the weight vectors. By gradually increasing the value of γ during the training process, the input vectors are drawn apart and eventually, when γ reaches 1, become the original input values.

The weight vectors will follow the preprocessed input vectors and classify different but similar input vectors as different. The convex combination method operates well, but slows down the training process as the weights are adjusting towards a moving target.

6.2 The Sequential Simulator

To facilitate the implementation of the parallel counterpropagation network simulation program, first a sequential version was built. The sequential version uses the parallel interface, but specifies only a master process on the transputer system. In this way it was experienced what was required to parallelize an existing sequential neural network simulation. Also the performance of the parallel simulators could be compared to the sequential version.

6.2.1 Design And Implementation

On basis of the information presented in the previous sections, the design and implementation of the simulation program was started. The design phase mainly consists of writing the program description file (see appendix A). The program description lays down what data structures and variables are to be used and how they will be organized in the implementation.

The presence of PREENS/CONVIS really helps a lot, as no effort has to be put in the design of the user interface or the modularization of the simulation program, which will normally take a lot of time.

To be able to use the parallel interface for a simulation running on a single transputer, a single processor topology had to be defined. The routines in the topology and data_distribution modules were coded as dummies or very shallow coded functions. No other adaptions were needed.

6.2.2 Results

The counterpropagation network simulation program was tested with a pattern set consisting of thirty-six patterns of one hundred binary pixels each, representing the twenty-six letters of the alphabet and the ten digits (See appendix B).

Varying the number of Kohonen neurons

The first test performed was the search for the optimal network size for this test pattern set. The network was initialized and trained using the random initiation method. Training per pattern was performed without monitoring any variable or data. The number of epochs was limited to 500, the error criterium set to 0.001. Figures 6.1 and 6.2 show the results of this test.

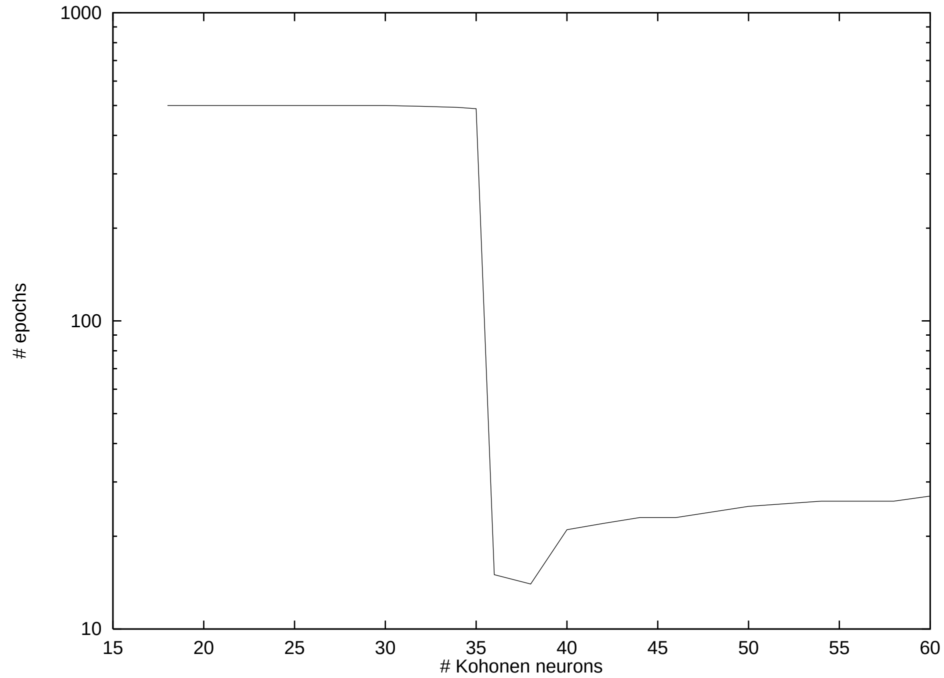


Figure 6.1: Number of epochs using different network sizes

It is not surprising that the optimum was found at thirty-six neurons in the Kohonen layer. In this case, a one-to-one relation will be built between the input patterns and the Kohonen neurons: each neuron will be activated for one unique pattern in the input set. With less neurons in the Kohonen map, some patterns will have to 'share' a neuron, which will result in worse network

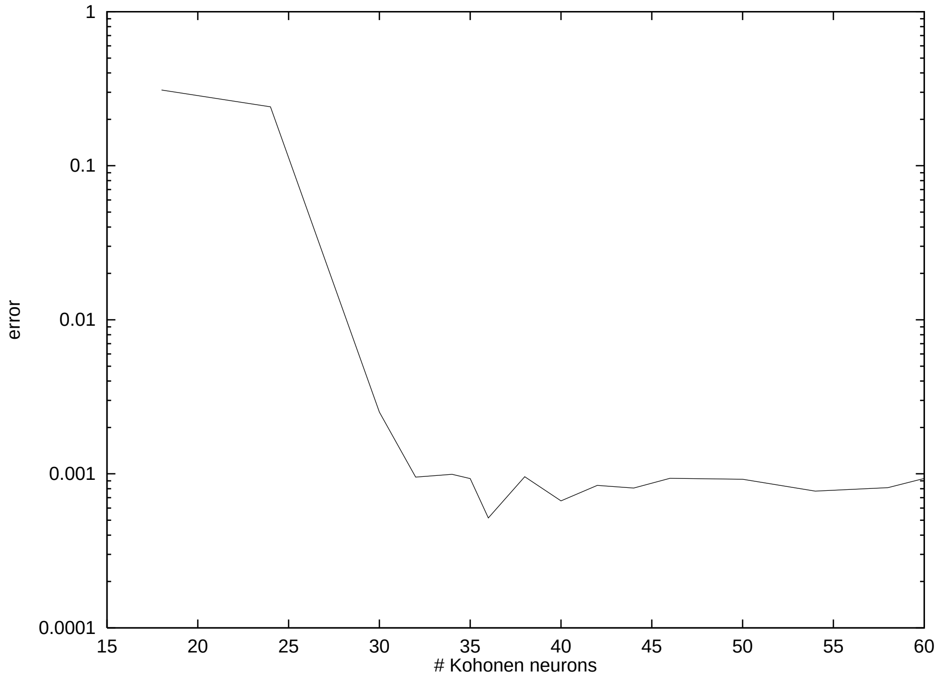


Figure 6.2: Network error after training using different network sizes

performance. It also shows that adding more neurons to the Kohonen map does not improve performance.

Training per epoch

With the optimal network size derived from the results above, another training session was started using training per epoch. It did not work. After a few epochs, the network error 'exploded' and reached for infinity. The cause of this effect was found in phenomena which can be called *resonance*.

Resonance occurs when the weights are over-adjusted. The process is shown in figure 6.3. Both the weight vector and the input vectors are depicted as points (a). The object of the training process is to adjust the weight (move the point) to make it resemble the input vectors. First, the error between the weight and input vectors is calculated (b). The error vectors are summed (c) and a part (α) is added to the weight vector (d). Because of this summation, the resulting weight vector will have a larger distance to the input vectors than the original one. During the training process, the weights will thus move away from the input vectors causing the error explosion.

With the cause known, the remedy was also found: the errors per weight should not be applied directly to the weights, but be scaled first. In the final implementation, the error vector values are divided by the number of patterns in the training set, thus keeping the weight adjustments small enough to avoid the resonance effect. The major drawback of this method is that the weight

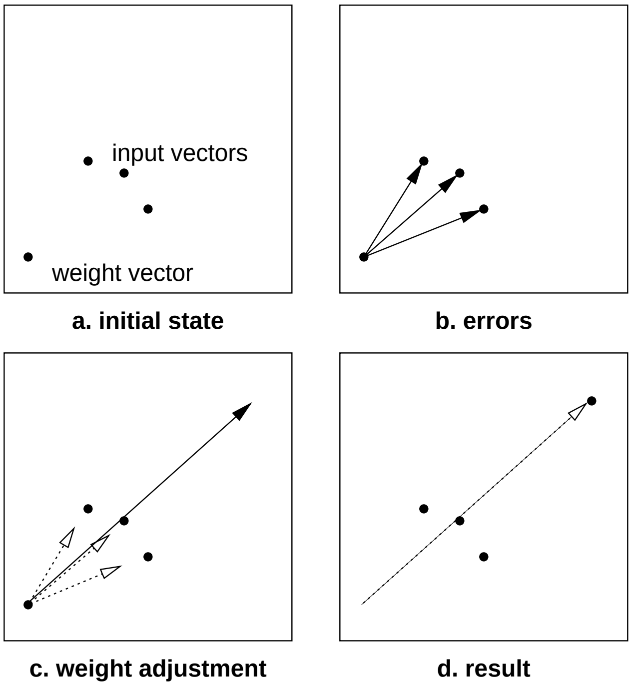


Figure 6.3: Resonance

adjustments are becoming very small, slowing down the training process. Figure 6.4 shows the error plots for training per pattern and per epoch using a network with thirty-six Kohonen neurons, with the error criterium set to 0.05.

The variations

The above tests were all done using accretive mode, one winner in the Kohonen layer, and using normal training. The variations, interpolative mode and the convex combination were tested also, but the differences in performance were minimal. Both [Hec90] and [Was89] state that improvement in performance is very problem dependent: *your mileage may vary...*

6.3 Inputset Parallelization

Once the sequential simulator was up and running, a start was made on the implementation of the parallel versions. The parallel simulator using input set decomposition was built first.

6.3.1 Design And Implementation

The design phase for the parallel simulators was almost non-existent, as the functionality of the programs was already known and implemented for the sequential version. To make the implementation process of the input set decom-

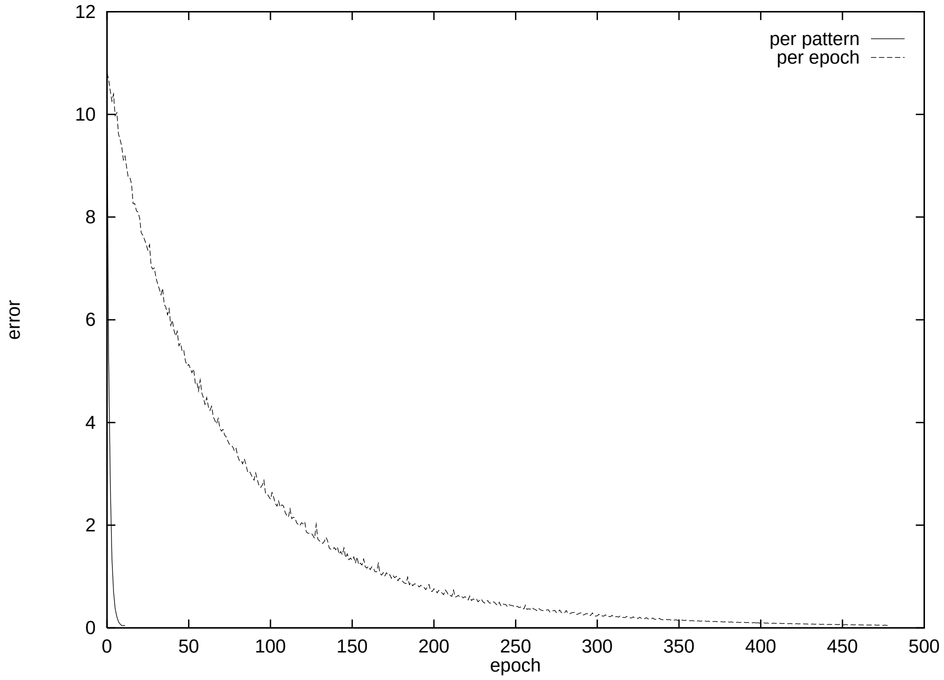


Figure 6.4: Error plots for training per pattern and per epoch

position easier, a dedicated implementation of the module `patterns` was built. This module takes care of the allocating, distributing and loading of patterns in a topology—a tree in this case.

The pattern loading and distributing routines were based on the routines in the geometrical data distribution module `tree_geom` (See section 5.3.3). The main reasons that this module could not be used directly is that patterns are treated different from other CONVIS datastructures and that typical pattern files contain two intertwined datastructures: the input patterns and the belonging output patterns.

Because the `patterns` module took care of the loading *and* the distribution of the patterns, no drastic changes were needed in the source of the simulation program. Apart from the adaptions discussed in chapter 2 only one extension had to be implemented: A routine `GatherMax` was written that selects the maximum of a set of values offered by the processors. This routine was needed for the calculation of the winning neuron(s) in the Kohonen layer.

6.3.2 Results

The input set parallel simulator was tested with the network sizes derived from the results obtained with the sequential version. All tests were performed on a neural network with thirty-six Kohonen neurons, using normal training and the error criterium set to 0.05. The network error was monitored using the standard monitoring tool included in PREENS.

Training per pattern and per epoch

The results of the first series of tests performed with the simulator are shown in figure 6.5. At a topology size of ten transputers the resonance effect occurs again. The cause of this lies in the fact that more patterns are evaluated at once and the situation thus starts to resemble that of the training per epoch using the sequential simulator. Again, the solution was to apply a scaling factor to the network. As scaling factor the number of transputers in the topology was used, as this equals the number of changes applied to the neural network at once.

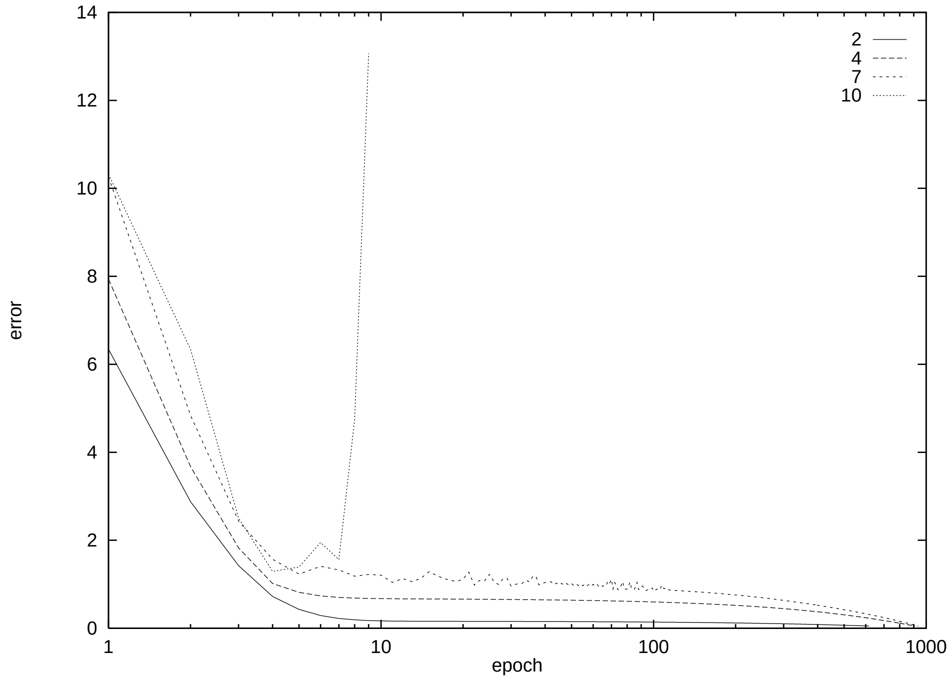


Figure 6.5: Training per pattern without scaling

With this additional adaption to the simulator, the testing was started again. Figure 6.6 show the error plots for a number of topology sizes. It clearly shows that the behavior of the neural network simulation 'moves' from the original training per pattern behavior of the sequential simulator towards the training per epoch situation.

Timing

For both training per pattern and training per epoch, the time used for calculation and communication was measured on the master processor. Two types of communication exist on the master processor: communication with CONVIS and the PREENS tools, and communication within the transputer topology.

Figure 6.7 shows the results for training per pattern. The most striking features in the diagram are the big drop in both communication and calculation

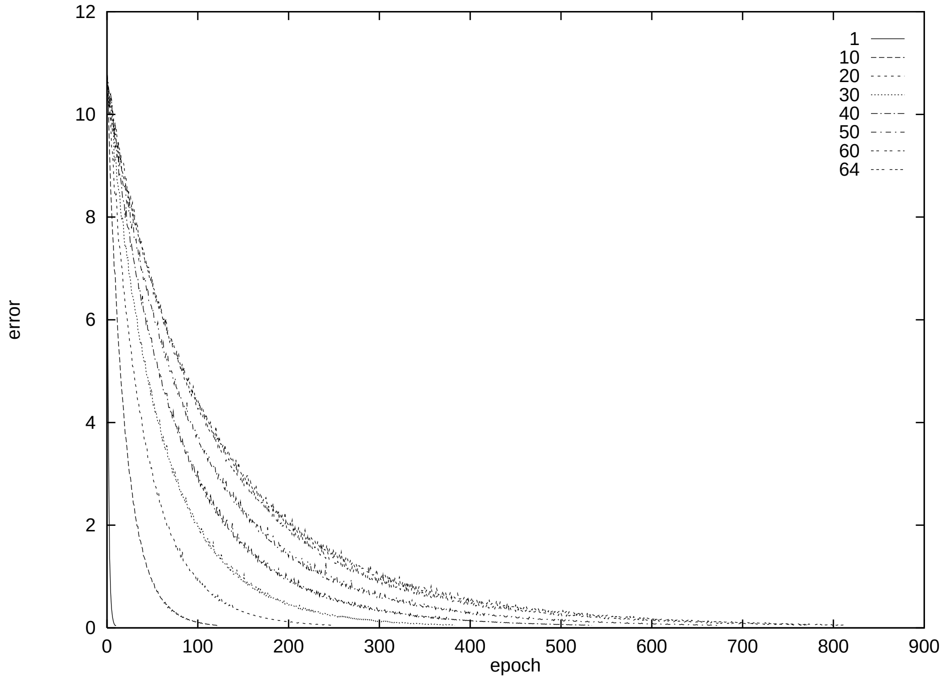


Figure 6.6: Error plots for several topology sizes

time at a topology size of thirty-six transputers and the fact that with the increasing topology size, also the calculation time increases.

The last feature can be explained by the fact that the more processors are added, the more epochs are required for the training, thus extending the calculation time.

The cause of the big drop lies in the combination of the size of the training pattern set, the distribution method and the training algorithm. The training set consists of thirty-six patterns, which are distributed over the processors in the topology. At a topology size of thirty-five transputers, each slave transputer will hold exactly one pattern, while the master processor holds two. This implies that each epoch contains two network updates: the first for all patterns on the slaves and one of the patterns on the master, the second just for the other pattern on the master.

At a topology size of thirty-six transputers, *all* processors will have one pattern, so each epoch will contain only one network update. Hence, both communication and calculation time will drop with nearly fifty percent. The same arguments hold for the smaller drops in the diagram (at nine and eighteen processors and below ten).

In this case,the optimal topology size for training per patterns seems to be a topology size of one processor, using no parallelism at all!

The diagram in figure 6.8 shows the calculation and communication time for training per epoch. The diagrams here are less surprising.

The computation time decreases in a $1/x$ ratio as the number of patterns per

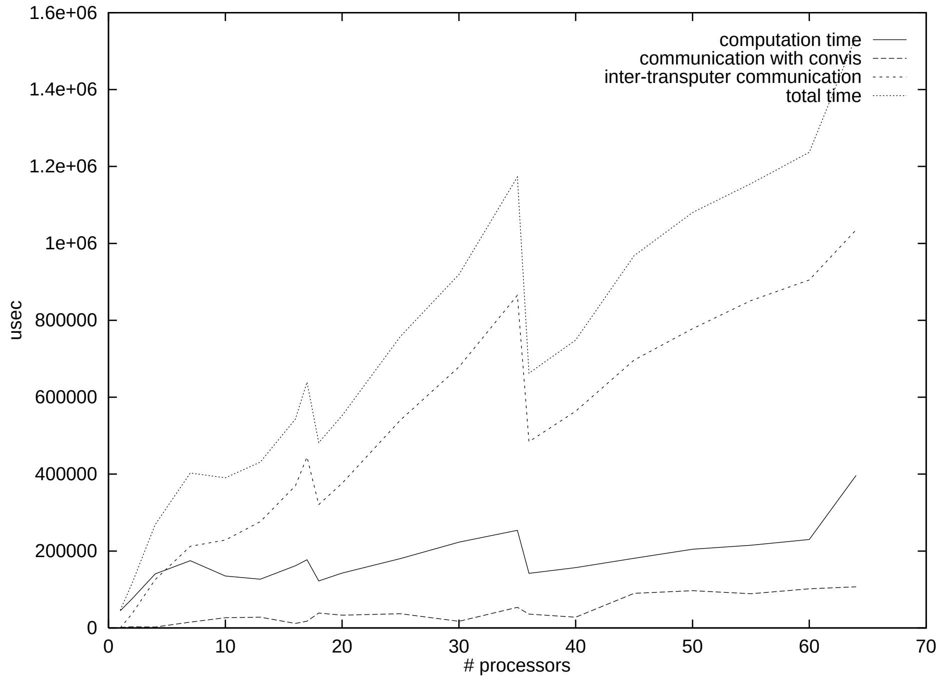


Figure 6.7: Timings for training per pattern

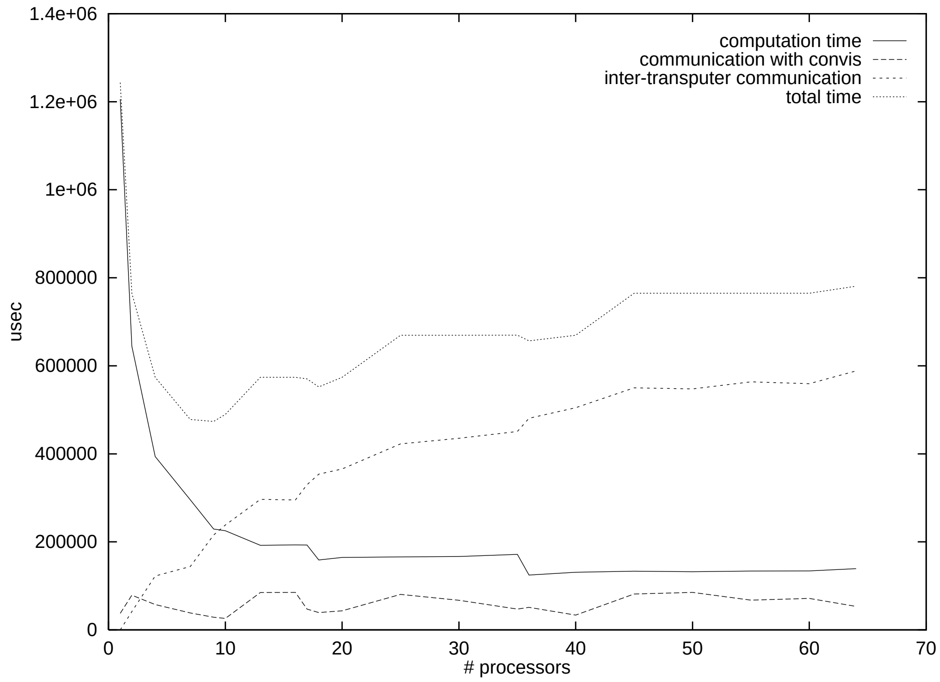


Figure 6.8: Timings for training per epoch

processor decreases. The inter-transputer communication time relates directly to the depth of the transputer tree and consequently increases in a $\log x$ ratio with the number of processors in the topology. The communication time with CONVIS should be almost constant, but fluctuates due to variations in work load of the CONVIS host machine. The optimal topology size in this case for training per epoch is nine processors.

6.4 Network Decomposition

The last simulation program built was the simulator using network decomposition. It was implemented on basis of source code of the sequential simulation program.

6.4.1 Design And Implementation

The main design issue for the network decomposition parallel was to determine which data structures would be distributed besides the weight and error matrices. Distributing the activation vectors, too, would decrease the per processor memory consumption, but it would make the implementation more complex. The inter-transputer communication would also increase as each layer requires the complete activation vector of the previous layer. Therefore, it was decided not to distribute the activation vectors and only distribute the weight and error matrices.

The implementation of the network decomposition parallel simulator was even easier than the implementation of the input set parallel variant. The distribution and allocation process for the distributed data is completely implemented in the modules datastructures and distribution. The result of this is that a minimal adaptions were needed in the source code of the neural network simulation program.

6.4.2 Results

The network decomposition parallel simulation program was tested with the same network size and parameters used with the input set parallel simulator. In contrast to the input set parallel simulation program, no surprises were encountered. Communication time with CONVIS remains constant, apart from the fluctuations due to the changing work load of the CONVIS host. The calculation time drops in a $1/x$ ratio and the inter-transputer communications grows in a $\log x$ ratio, as expected. In both training methods, the optimal topology tree size appears to be the tree using nine or ten transputers.

6.5 A Comparison

Training per pattern is, in this case, much more efficient than training per epoch—about 30 times faster! The main cause of this lies in the scaling performed to suppress resonance. If a more efficient resonance suppressing technique could be applied, training per epoch should become more efficient.

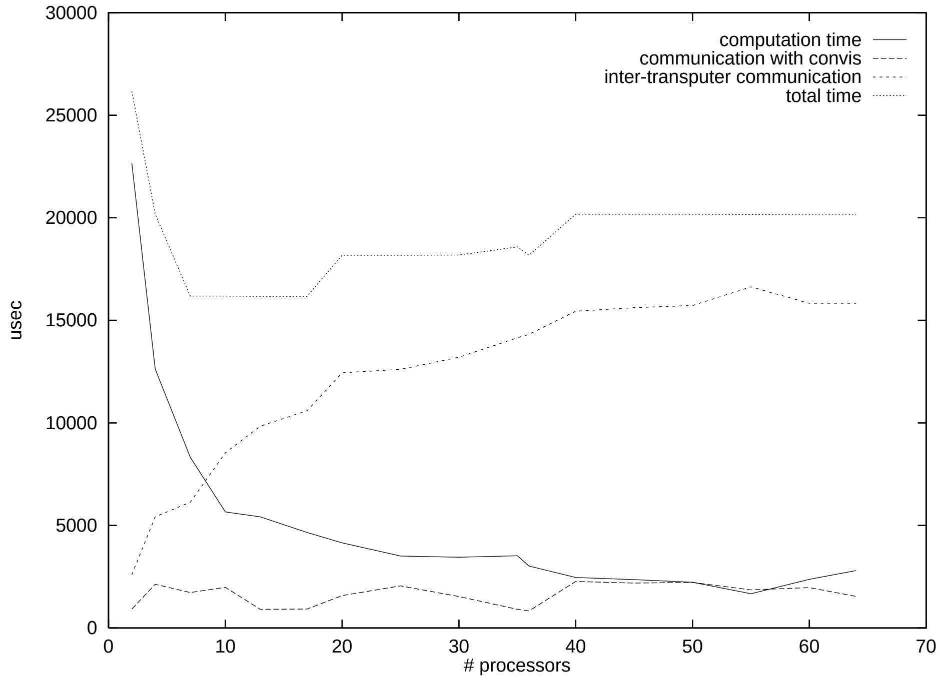


Figure 6.9: Timings for training per pattern

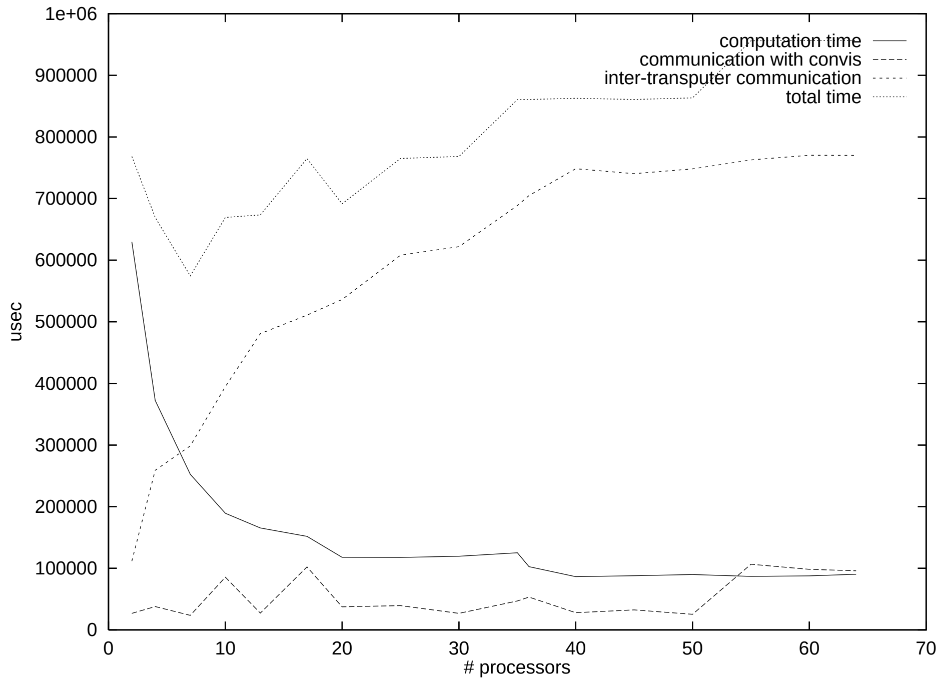


Figure 6.10: Timings for training per epoch

Comparing network decomposition with input set decomposition is more difficult. For input set decomposition, the behavior of the simulation program changes with a changing topology size, which makes this training method/parallelization combination stand out of the rest of the combinations. The other three combinations behave in much the same way, all having a topology with nine transputers as an optimum. Most efficient training method/parallelization combination will then become training per pattern using network decomposition.

Chapter 7

Conclusions

The final chapter of a thesis always carries the conclusions. This thesis is not an exception to that rule. Conclusions will be presented concerning PREENS and CONVIS in general, the parallel interface and the parallel counterpropagation network simulation program.

7.1 PREENS And CONVIS

The concept of PREENS and CONVIS is clearly well thought of. A consistent user interface, which can be tailored using the program description, is provided for the user; the programmer can use a complete simulation program template, in which he only has to fill in his neural network simulation procedures.

However, there are certain flaws and peculiarities in the implementation. Eg., CONVIS does not respond well when a monitoring tool is closed down 'in action': four or five windows pop up panicing about a lost communication after which CONVIS exits.

The messages sent between the simulation program and CONVIS could be used far more efficiently. Eg., when a variable is monitored, only it's value is important for the monitoring tool. This value could be put in the `value` field of the corresponding message. However, each time the variable is sent to the monitoring tool, not only its value, but also all other information on that variable is sent (name, min and max values, type, etc).

7.2 The Parallel Interface

The parallel interface serves its purpose well. Almost no effort had to be put in the parallelization of the simulation programs. There seems to be no difference in 'external behavior' between the original interface and the parallel version.

Some of the flaws that were in the original interface will, of course, also be found in the parallel version, as CONVIS relies on certain behavior of the interface. Changing the monitoring behavior, for example, would also imply a re-implementation of CONVIS and *that* was out of the scope of this project.

Some extensions, however, would improve the usability of PREENS for very large neural network simulation programs. The current interface is implemented on hardware, which is nowadays not anymore state-of-the-art. Porting the interface to another platform should not be very difficult: mainly the modules transputer and topology need re-implementation to make them tailored for a new platform.

The topology module implemented used a ternary tree. For other topologies, suitable modules could be made. However, this is not the most flexible way – an auto-configuring topology module would be the most versatile solution.

7.3 The Counterpropagation Network Simulator

Initially, the counterpropagation network simulation programs were just implemented to test the parallel interface. However, the results obtained indicate that further research on this type of neural network would be appropriate.

First of all, the existing simulation programs could be further evaluated using different training pattern sets, preferably larger than the pattern set used now. The fact that the training set used could be trained in fifteen epochs suggests that this set is fairly 'easy'.

Secondly, an alternative to the scaling method (to suppress resonance) should be found. [Hec90] suggests that the counterpropagation network performs best when the patterns used are scaled to unit length. As the goal of the training process is to make the Kohonen weight vectors resemble the input vectors, resonance could be avoided by scaling the weight vectors to unit length at each network update. Then, a more resultive comparison between the training method/parallelization combinations can be made.

Also, the impact of the communication between the simulation program and CONVIS was kept minimal in all test cases. Only one variable (the network error) was monitored. The implementation of the parallel interface presumes that the value of a variable (a variable that is registered in the action database, that is) is the same on all processors and will thus 'just' send the value of the variable on the master processor. When distributed data is monitored, however, the data will have to be gathered from all processors, which will influence both the communication time between the simulation programand CONVIS, and the inter-transputer communication time.

Appendix A

Program Description Of The Counterpropagation Network

```
# counter propagation network description

7 0

action "load network" { 0
  parameter { "filename"        file     7 0 0 T "network" }
  parameter { "nr of inputs"    int      100    1  MAX T "" }
  parameter { "nr of Kohonen"   int     1000    1  MAX T "" }
  parameter { "nr of Grossberg" int      100    1  MAX T "" }
  parameter { "nr of K winners" int        1    1  MAX T "" }
  parameter { "weights min"     double  0.1 -1.0  1.0 T "" }
  parameter { "weights max"     double  0.9 -1.0  1.0 T "" }
  data      { "inputs"          double []            F "" }
  data      { "K weights"       double []            F "" }
  data      { "K activations"   double []            F "" }
  data      { "G weights"       double []            F "" }
  data      { "G activations"   double []            F "" }
  option    { "load"        2 0 "new" "from file" }
  option    { "initialize"  2 0 "random" "convex combination" }
}


action "save network" { 0
  parameter { "filename"        file    7 0 0 T "network" }
}


action "load patterns" { 0
  parameter { "filename"        file   12 0 0 T "patterns" }
  parameter { "default off"     double  0.1 -1.0  1.0 T "" }
  parameter { "default on"      double  0.9  0.0  1.0 T "" }
```

```
  data      { "input patterns"  double []            F "" }
  data      { "target patterns" double []            F "" }
  option    { "patterns for" 3 0 "training" "recall" "classify" }
  setting   { "scale patterns" OFF }
}


action "save patterns" { 0
  parameter { "filename"        file    7 0 0 T "patterns" }
  option    { "patterns for" 3 0 "training" "recall" "classify" }
}


action "learn" { 1
  parameter { "nr of epochs"    int    1000    1  MAX T "" }
  parameter { "error criterium" double 0.05  0.0  1.0 T "" }
  variable  { "epoch"           int       0    0  MAX T "epoch" }
  variable  { "K learning rate" double  0.7  0.0  1.0 T "epoch" }
  variable  { "G learning rate" double  0.5  0.0  1.0 T "epoch" }
  variable  { "ccm ratio"       double  0.0  0.0  1.0 T "epoch" }
  variable  { "error"           double  1.0  0.0 10.0 T "epoch" }
  data      { "inputs"          double []             T "epoch" }
  data      { "K weights"       double []             T "epoch" }
  data      { "K errors"        double []             T "epoch" }
  data      { "K activations"   double []             T "epoch" }
  data      { "G weights"       double []             T "epoch" }
  data      { "G errors"        double []             T "epoch" }
  data      { "G activations"   double []             T "epoch" }
  data      { "input patterns"  double []             T "epoch" }
  data      { "target patterns" double []             T "epoch" }
  data      { "confusion matrix" double []            T "epoch" }
  option    { "train per"    2 1 "pattern" "epoch" }
  option    { "train mode"   2 0 "normal" "convex combination" }
  setting   { "make confusion matrix" OFF }
}


action "recall"  { 1
  parameter { "save file"       file      7    0    0 T "cpn.recall" }
  data      { "input patterns"  double []            F "" }
  data      { "target patterns" double []            F "" }
  data      { "classification"  double []            T "" }
  data      { "confusion matrix" double []            T "" }
  option    { "recall mode"  2 0 "net output" "classify" }
  setting   { "make confusion matrix" ON }
}
```

```
action "classify" { 1
  parameter { "save file"       file      7    0    0 T "cpn.classify" }
  data      { "input patterns"  double []             F "" }
  data      { "classification"  double []             T "" }
  option    { "mode"         2 1 "net output" "classify" }
}
```

Appendix B

Patterns Used For Training

Figure B.1 shows the patterns used for training the counterpropagation network descibed in chapter 6.

Figure B.1: Patterns used for training the counterpropagation network

Bibliography

[Dav91] Davies, I. *The Helios Parallel Operating System* – Hertfordshire : Prentice Hall International (UK) Ltd., 1991.

[Heb49] Hebb, D. *The Organization Of Behaviour* – New York : Wiley, 1949.

[Hec90] Hecht-Nielsen, R. *Neurocomputing* – New York [etc] : Addison-Wesley Publishing Company, 1990.

[Koh84] Kohonen, T. *Self-Organization And Associative Memory* – Berlin [etc] : Springer-Verlag, 1984.

[McC43] McCullogh and W. Pitts, W.S. "A Logical Calculus Of The Ideas Immanent In Nervous Activity", *Bulletin of Math. Bio*, **5**, 1943, pp. 115–133.

[Min69] Minsky and S. Papert, M. *Perceptrons* – Cambridge MA : MIT Press, 1969.

[Qui94] Quinn, M.J. *Parallel Computing: Theory and Practice* – New York [etc] : McGraw-Hill, Inc., 1994.

[Rum86] Rumelhardt, G.E. Hinton and R.J. Williams, D.E. *Learning Internal Representations By Error Propagation* in Rumelhart and J.L. McClellandi, D.E. [Eds.], **Parallel Distributed Programming: Explorations in the Microstructure of Cognition 1** – Cambridge MA : MIT Press, 1986, pp. 318–362.

[Sch91] Schouten and drs. L.G. Vuurpijl, dr. Th.E. *Transparanten en artikelen (deel 3) - Collegedictaat Neurale Netwerken - Voorjaar 1991* – Nijmegen : University of Nijmegen, 1991.

[Vee90] Veer, B. *The CDL Guide* – Bristol : Distributed Software Ltd., 1990.

[Vuu94] Vuurpijl, dr. Th.E. Shouten and J. Vytopil, drs. L.G. *CONVIS: Action Oriented Control and Visualization of Neural Networks - Introduction and Technical Description* – Nijmegen : University of Nijmegen, 1994.

[Was89] Wassermann, P.D. *Neural Computing, Theory and Practice* – New York : van Nostrand Reinhold, 1989.

[Wer74] Werbos, P.J. *Beyond Recognition, New Tools For Prediction And Analysis In The Behavioural Sciences* – PhD thesis Harvard University, 1974.

[Wir76] Wirth, N. *Algorithms + Data Structures = Programs* – Englewood Cliffs, N.J. : Prentice-Hall, 1976.