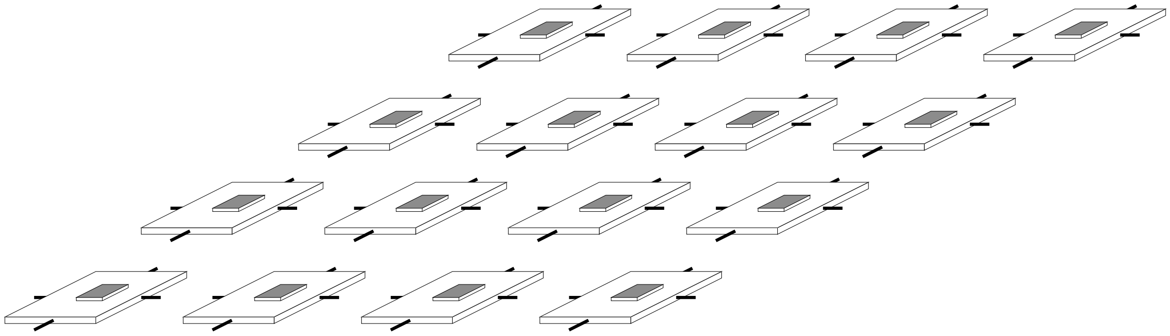
Platforms for artificial neural networks

Neurosimulators

and

Performance prediction of MIMD-Parallel Systems



Louis Vuurpijl

Platforms for artificial neural networks

Neurosimulators

and

Performance prediction of MIMD-Parallel Systems

een wetenschappelijke proeve op het gebied
van de Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
vrijdag 6 februari 1998
des namiddags om 3.30 uur precies

door

Louis Gerard Vuurpijl

geboren op 4 april 1964 te Gorinchem

Promotor: Prof. Dr. Ir. Jan Vytopil

Co-promotor: Dr. Th. E. Schouten

Manuscriptcommissie:

Prof. Dr. Ir. F.C.A. Groen (UvA)

Prof. Dr. P.T.W. Hudson (RUL)

Dr. L.R.B. Schomaker

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Vuurpijl, Louis Gerard

Platforms for artificial neural networks —
Neurosimulators and Performance prediction of
MIMD-parallel systems /
Louis Vuurpijl.
Proefschrift Nijmegen. - Met lit. opg. - Met
samenvatting in het Nederlands
ISBN 90-9011347-9
NUGI 831
Tref. neurale netwerken / computertechniek

Voor Ron

Contents

Preface

When I started my PhD project and talked to Ron about all the things I would like to examine, and all the things that I would discover, he immediately reacted very enthusiastically. It was how he always reacted, and as he was also doing his PhD, we planned to work together on a paper about neural networks and satellite image classification. Though he was working at the Joint Research Center in Italy, we were always in contact. By phone and through The Internet via email or using `talk`. We had the same promotor and supervisor, so Ron often had to come to Nijmegen to discuss chapters of his thesis with Jan and Theo. On such occasions, we would go out to bars like de Kluis en de Fuik to meet the numerous friends he had here in Holland. One time Jan Vytopil, Theo Schouten, Harry Duys and I visited the JRC, where we met Ron and his supervisors Graeme Wilkinson and Jacques Stakenborg. It was a great visit, because in the weekend we went into the mountains to go skiing in Cervina and Zermatt, near the Matterhorn.

I will never ski with Ron again, because he tragically died in a car accident on August 1 in 1995. It was about one month before he would get his PhD title, which he eventually got posthumous. As always, Ron would reach his goals, so we succeeded in finishing our joint paper. In the summer of 1995, Ron presented our paper. It was the first and also the last paper of many others we had planned. This thesis is dedicated to Ron Schoenmakers. Ron jonguh, bedankt!

In fact, the paper I wrote with Ron was only part of the work I would have to do. In January 1990, the PREENS project (Parallel Research Execution Environment for Neural Systems) was launched. It was part of the Dutch national SPIN project (Stimulerings Plan Informatica Nederland) and SNN (Stichting Neurale Netwerken), a collaboration of four universities and several Dutch commercial firms. The project was also funded by the Esprit Parallel Computing Action (PCA 4106). Fundings were available for a period of four man years and made it possible to acquire a 64 multi-transputer parallel processor system. The project was carried out at the Department of Experimental Informatics for Technical Applications, supervised by Professor Jan Vytopil and guided by Dr. Theo Schouten.

Within the PREENS project, the goal was to examine the suitability of transputer systems as an execution platform for artificial neural network simulations. Another goal was to develop an execution environment for simulating neural networks, an environment which is also known as *neurosimulator*. This thesis will take the reader to several aspects from the world of neurocomputing. In particular, it will explain about how MIMD-parallel computers (like transputer systems) can be used for simulating neural networks, and how such systems can be evaluated in terms of the performance, speedup, scalability and efficiency that can be achieved. The reader will become acquainted with neurosimulators, the world of neurocomputing, and I hope to explain the design principles of the neurosimulator PREENS I developed.

The first years of my research I shared my office with Peter Snel, whom I started to like because of his nice character. In those days, everybody at our department had an Apple Macintosh on his desk and Peter was 'the mac-man' (of course, the rest of my life I will use Unix workstations). We had a nice group of people working together, and had regular coffee-meetings with Jan, Theo, Peter, Mirese, Martin, Hanno and also a lot of graduate students. I would like to thank them all for being such a pleasant company. And also for helping and stimulating me with my work. Especially Theo Schouten, my supervisor. Not only could I always enter his room, for a chat or some serious stuff, he also invited me at home to enjoy the fabulous company and cooking of his wife Riet. And you too Jan, for helping me out with my "vervangende dienst", offering me a job as AiO, and guiding me through the years of my research and writing.

Though now I am an ultra-experienced programmer and Unix-user, by the time I started my research I knew nothing about the ins-and-outs of C. Fortunately, the "student room" was next to mine and I could always drop in to come up with famous questions like "Why doesn't this do what I want it to do?"

```
void make_hello_world (char *ptr)
{
     ptr = strdup("Hello world!");
}

int main (int argc, char *argv[])
{
     char *hello;

     make_hello_world(hello);
     printf ("%s\n",hello);
}
```

I guided a number of students along their (graduate) projects, whom I would like to thank for being there, doing part of my work and solving stupid questions like the one above. Albertr, Reinoud, Tricky, Silvio, Hans, Rons, Ronl, Parcival, Eric, Rene, Charles, Chris, Albertk, Albertb, Paul, Jan Willem, . . . , thank you all.

The last years of my PhD research, I shared my room with Maurice klein Gebbinck. I started to know and like Maurice as one of the nicest people in the world. We developed our own communication methods, and the two of us must have been a terrible nuisance for anyone sitting in a room closer than 100 meters. We sang songs on Monday morning, having heard them at “De Hollandse Avond in De Fuik” on Sundays. If I tapped the rhythm of a song with my fingers, Maurice would join me. Or vice versa. And of course we always spoke to our computers, cursing because they always respond too slow, something which really became apparent with the advent of The Internet and WWW. Maurice, bedankt en maak het maar snel af, je promotie.

Since 1995, I am working with the group of Lambert Schomaker on a seemingly totally different subject, handwriting recognition. He also helped me finishing my thesis. He used to let me take days off for working on it (and this year even two months). Lambert, bedankt voor al je adviezen en wijze lessen. At the NICI, I shared my room with Janek Mackowiak. Janek, thanks for the cooperation and you being there.

Despite of the existence of the “Verbond van Slechte Mannen” (the alliance of bad men), of which Ron is honorary member, I managed to finish this thesis. Ron, Olaf, Jack, Steef, Marc, Johan, Joost en ook Renze, bedankt voor jullie vriendschap de afgelopen jaren. Zonder jullie was het veel eerder afgekomen.

Last but not least, I would like to thank my brother Gerrit and my mother Truus for always supporting and believing in me. Now, they are proud of me. I always was proud of the two of you, and I always will.

1

Introduction

Outline

In this thesis, two platforms for simulating artificial neural networks are discussed:

- ◇ MIMD-parallel processor systems as an execution platform
- ◇ neurosimulators as a research and development platform

Because of the parallelism encountered in neural networks, distributed processor systems seem to provide a proper underlying execution platform. The suitability of the class of MIMD-parallel computer platforms (in particular multi-transputer systems) for neural network simulation programs is discussed in this thesis. In order to evaluate the suitability of such systems, a new performance prediction method is presented. An introduction to the chapters discussing this method is given in this chapter.

Neurosimulators provide a platform for simulating, developing, evaluating and executing neural network models. In the last two chapters of this thesis, neurosimulators are examined: environments for the development and simulation of artificial neural networks. By considering their common features, and the requirements of their users, the design criteria for a new neurosimulator are specified. The design, implementation and evaluation of PREENS, an action-oriented neurosimulator is presented in the final chapter.

Parallelism and parallel neural network simulations

The occurrence of parallelism can be observed in all aspects of life. For instance, all creatures live their lives simultaneously. Often, they also cooperate together to accomplish one or more tasks, like ants building an ant hill, or lions chasing a prey. In production plants, many machines may operate simultaneously on identical tasks, distinct tasks, or parts of tasks. Parallelism can be observed in the construction of houses or roads, in cars and pedestrians at a busy junction, and even in tasks as doing household chores.

Artificial neural networks (ANNs) also feature parallelism, on several levels of detail. They contain a large number of processing elements, *neurons*, connected via an even larger number of *weights* (modeling axons and synapses). In biology, each neuron, axon and synapse operates in parallel. At a more coarse grained level of detail, multiple modules or layers of neurons and weights can be identified, all operating simultaneously. In Chapter 2, a brief introduction into the topic of neural networks is given. That chapter also serves as a further introduction to this thesis, introducing the concept of neurosimulators and of performance modeling for the class of multiple instruction, multiple data (MIMD) parallel computer systems.

The goal of exploiting parallel computing is to finish a task in a smaller amount of time, or to handle more (distinct) tasks in the same amount of time. In Chapter 3, the parallelism encountered in computer systems is discussed. Coarse grained multi-processor systems are for example networks of workstations or transputer systems. On the other hand, computer systems like the connection machine, or array processors, feature parallelism at a finer level of detail.

When considering the parallelism occurring in neural networks, parallel processor systems seem to provide a natural underlying hardware platform for implementing them. The advantage of having such execution platforms is that parallel implementations run faster than sequential ones. Disadvantages are that parallel implementations are harder to program and more dedicated to one specific neural network model or application. A large amount of parallel implementations of neural networks on several levels of detail are reported in the literature. The lowest level represents a one-to-one mapping of weights and neurons onto analog or digital circuits. An overview over such implementations can be found in [45, 79, 103]. A wide range of other parallel execution platforms like the Connection Machine [94], GF11 [124], MasPar [17] and transputer systems [81, 82, 84, 104, 107] are proposed for higher (more abstract) levels of parallelism. These machines can be massively parallel, i.e. contain a large number of relatively simple parallel processors ($\gg 1000$), or they can consist of a smaller number of more general purpose nodes.

In Chapter 3, multi-transputer systems are discussed. They are used as an execution platform for parallel neural network simulations in this thesis. The main reason why transputers were chosen for this PhD study are that the transputer is a general purpose computer, making it suitable for a large variety of neural network models. Whereas massively parallel systems contain many nodes, the number of nodes for MIMD-processor systems is limited (in practice < 1000). As a consequence, the number of neurons and weights exceeds the number of processors. So when implementing a neural network on a generic MIMD-execution platform, the neural network model has to be decomposed somehow over the available processors, where groups of neurons and weights are placed on different processors. Techniques for implementing neural networks on MIMD-processor systems like the transputer are described in, e.g., [38, 100, 110]. These techniques are discussed in detail in Chapters 5, 6 and 7. Two issues are of importance for parallel neural network decompositions: a) make sure that each processor has an equal amount of work to do (*load balance*), and b) make sure that the amount of synchronization and communication overheads is kept as low as possible.

Suitability of MIMD-processor platforms for ANNs

A large part of this thesis is dedicated to a method for determining whether an execution platform is well-suited for a certain application. For such an evaluation, several questions can be raised [113, 114]. The first question is what execution performance can be achieved for a given machine configuration and type and dimension of application. The second question is whether by increasing the amount of computing and communication resources, the total execution time for a given application can be reduced, i.e., what level of speedup can be achieved. A third question concerns the scalability of the platform and application, i.e., does the execution time stay constant if the size of both the application and platform is scaled up? In Chapter 4, a method is introduced for predicting the performance of MIMD-processor systems for ANNs. For a given processor architecture, the method models the calculation time required for executing the application, and it models the time required for communication. Based on measurements of the calculation time on one processor and the communication time between two processors, predictions can be made for larger processor systems. Using this method, the three questions issued above can be answered.

As examined in [110, 113], parallel neural network simulations require several kinds of communication. In Chapter 5, a communication layer implementing the typical communication requirements of distributed neural network implementations is presented. A model for the required communication time is introduced and evaluated for three different transputer systems. The first is the Nijmegen Super Cluster, a system containing 64 T800 transputers. The second is the GCEL-512, containing 512 T805 processors. And the third is the PowerXPlorer, a system containing 32 PowerPC 601 nodes. The latter two machines are located at the University of Amsterdam.

In Chapters 6 and 7, two different decomposition techniques are described: data set decomposition and network decomposition. Both techniques are applied on two popular neural network models, the multi-layered perceptron [83] and the Kohonen self-organizing map [58]. The parallel implementations for both neural networks use the communication layer introduced in Chapter 5. The performance prediction method discussed in Chapter 4 is evaluated both for dataset decomposition and network decomposition techniques. It will be pointed out that using the method, the suitability of the class of MIMD-parallel processor systems for artificial neural network simulations can be predicted accurately.

Neurosimulators

The final two chapters of this thesis are concerned with platforms dedicated to artificial neural networks simulations, called neurosimulators. A neurosimulator is defined as a set of software and/or hardware components that can operate together to support the construction, manipulation, visualization or (fast) execution of neural network simulations [116, 112]. This thesis presents a new kind of neurosimulator, called PREENS, a parallel execution environment for neural systems.

In Chapter 2, an introduction is made to the world of neurocomputing. The different user groups involved in using, developing, or implementing neural networks are identified. Furthermore, the neurocomputing life-cycle is presented. This life-cycle contains four phases: Initiation, Tuning, Testing, and Operation. In the Initiation phase, the task to be performed and its preconditions are determined. In the Tuning phase, a chosen neural network model is tailored for the task determined in the initiation phase. In the Testing phase, the performance of the resulting neural network is evaluated. Performance in this context may involve execution speed, recognition accuracy, or reliability. In the final phase (Operation), the resulting optimized and tested neural network is used in an actual application.

Existing neurosimulators and the features they exhibit are reviewed in Chapter 8. As distinguished by Recce *et al* [80], neurosimulators can be distinguished in application-oriented, algorithm-oriented, and general programming systems. Features they may share are a graphical user-interface, an algorithm library containing a set of implemented neural network models, support for building new models, application specific tools, dedicated hardware accelerators, etcetera.

Based on the observations made when considering these features and based on the requirements from users in the world of neurocomputing, in the final chapter of this thesis (Chapter 9), the design and implementation of PREENS is presented. PREENS comprises a neural network algorithm library, a set of tools, and a manager called CONVIS for controlling tools and simulation programs. Tools, CONVIS and a simulation program can run as separate processes in a heterogeneous computer network. This, and a new concept called action-oriented program descriptions form the main differences between PREENS and existing neurosimulators.

The concept of actions and their associated components is explained in Chapter 9. Based on this, a set of interface definitions is specified via which new tools or neural network simulation programs can be integrated in PREENS relatively easy. PREENS will be evaluated on a real-world application, the classification of remotely sensed (satellite) images.

2

Platforms for artificial neural networks

Outline

This chapter serves as an introduction to two concepts: a) performance prediction of MIMD-parallel execution platforms for artificial neural networks, and b) the world of neurocomputing and neurosimulators.

Artificial neural networks can be considered as computer programs inspired by the processes taking place in the human nervous system. In this chapter, a brief introduction to artificial neural networks is given. Because of the parallelism observed in biological neural networks, parallel processor systems like the transputer are believed to provide a natural and efficient platform for running artificial neural networks. The suitability of such a platform can be expressed in terms of the execution performance, speedup and scalability that can be achieved. A method for predicting these parameters for parallel neural network simulations is introduced in this chapter.

Neurosimulators are a collection of software and hardware tools for simulating artificial neural networks. They provide a platform for users involved in applying or developing neural networks: users "doing neurocomputing". In this chapter, the world of neurocomputing, a taxonomy of users doing neurocomputing, and the neurocomputing life-cycle are presented. An introduction will be given towards the design of a new type of neurosimulator, called PREENS.

2.1 Artificial neural networks

We, human beings, are capable of performing difficult tasks seemingly effortless; tasks that are not handled very well by computers. For example recognition of speech and visual stimuli in complex scenes, performing sports, and our ability to constantly learn by experience are still very difficult to be performed by a machine. Since the beginning of the computer era, people have been wondering about how to incorporate knowledge of our brain and the nervous system into mathematical models and computer simulations, and how to use these biologically inspired models for a specific application area.

A large number of distinct neural network models exist, and within each model, there exist a large number of variations. Well-known neural network models are backpropagation or multi-layered perceptrons [83], Kohonen networks [58], Hopfield networks [49], counter-propagation networks [44], ART networks [13, 12, 14, 15], Boltzmann networks [92], etc. On a regular basis, new publications are appearing that report minor or significant changes in these paradigms, so the number of models is still increasing rapidly. However, all models share a number of important features: they exchange *inputs* and *outputs* (modeled by scalar numbers) with their environment, they consist of a large number of processing elements (*neurons*), they have *connections* between the neurons, neurons receive inputs from neighboring neurons, compute their state of *activation* based on this information, propagate the activation following an *activation function*, and they all have the ability to learn by changing their connections following a certain *learning mechanism*. A particular connection topology, activation mechanism and learning mechanism together form the basic ingredients of a particular artificial neural network model[1].

A general neuron model

Each neuron j has an activation value a_j and a threshold θ_j. Connections between two neurons i and j are modeled by a weight value w_{ij}. A neuron receives activation values from its input neurons, computes the weighted sum of its inputs and weights, and outputs an activation value which is computed by the activation function f_a.

$$a(j) \quad = \quad f_a(\sum w_{ij} \cdot a_i - \theta_j) \tag{2.1}$$

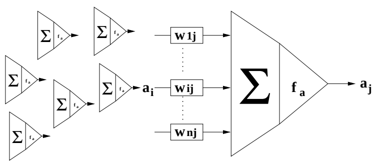


Figure 2.1: *A simple neuron model, receiving inputs a_i from other neurons.*

[1] For an overview over the different neural network paradigms, I refer to the frequently asked questions (FAQ) of the Usenet newsgroup `comp.ai.neural-nets`. It contains an up-to-date list of references to books and journals, with reviews and sectioning in, e.g. introductory, business, intermediate and advanced categories. An excellent book is *Neural networks for pattern recognition* by C.W. Bishop [8].

The activation function is usually of a simple form and can, for example, consist of a threshold function or a sigmoid:

$$\text{threshold} \qquad f_a(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{sigmoid} \qquad f_a = \frac{1}{1 + \exp(-x)}$$

Throughout this thesis, two popular artificial neural networks will be used as a running example, the backpropagation network [83] and the Kohonen self-organizing feature map SOM [58]. Details of their connection architecture, training algorithms and activation functions, and details of different implementations on sequential and parallel machines will be discussed when required in chapters 4, 6 and 7.

The Kohonen SOM

The Self-Organizing Feature Map (SOM) [58] is — besides backpropagation — the most well known neural network paradigm. The neural network is arranged in a N-dimensional grid of neurons (usually N=2), the feature map (see Figure 2.2). Each neuron is fully connected to an input layer which represents a feature vector of the input data.

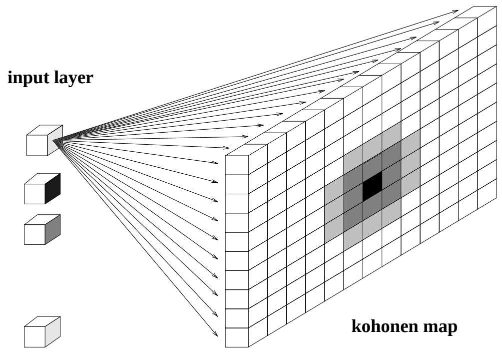


Figure 2.2: *Architecture of the self-organizing feature map. The winning (black) neuron is excited the most, the "bubble" of neurons in its neighborhood is also excited.*

As with most neural networks, the SOM distinguishes two phases, an activation (or recall) phase and a training phase. During activation, an input vector is clamped on the input layer and each neuron computes its match with the input. Two methods are often used, computing the Euclidean distance or taking the inner product. The latter requires that all weights and input data are normalized. The neuron with the best match is called the *winner* and its weights form an internal representation of the input feature. During training, the winning neuron and the neurons laying within its neighborhood update their weights. A complete dataset can be learned by the SOM by repeatedly clamping an input vector, activating the winning neuron and its neighbors and updating their weights.

The backpropagation network

The implementations of the backpropagation network described in this thesis all use the algorithm described by Rumelhart in [83]. Among all the different variations on the original algorithm, this is probably the most widely used. The backpropagation network has a layered architecture of one input layer, one or more hidden layers and one output layer.

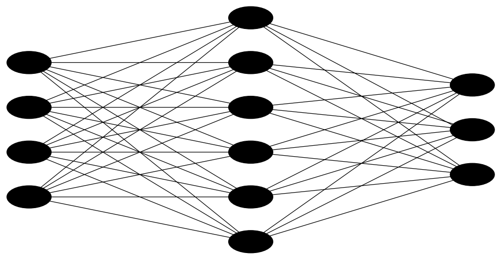


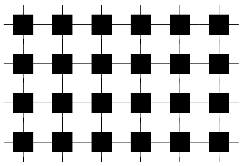
Figure 2.3: *Architecture of a three-layered 4x6x3 backpropagation network.*

Neurons in two subsequent layers are fully inter-connected. During activation of the network (the forward pass), information consisting of neuron activations flows through the connections from the input through the hidden layers to the output layer. Each input pattern is clamped on the input layer. The activation of a neuron in a hidden or output layer is computed based on the activation of its input neurons and the strength of the corresponding connections, as in Equation 2.1. During training (the backward pass), the information flow is reversed, and each neuron propagates the error values it has produced back. The error values are computed based on the difference between target patterns and computed outputs. Based on the error values, each neuron can compute the contribution of its input weights to the error, the so-called delta. Using its delta, learning rate and other parameters, each neuron updates its weight values.

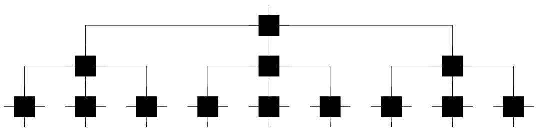
2.2 Neural networks and execution platforms

Since the increasing engagement of research and development institutions, and academic, industrial and commercial users in the area of neurocomputing, the demand for high performance implementations of neural networks has arisen. Requirements concerning the amount of data that has to be handled, the size of the neural networks involved and the response times that have to be fulfilled have become more and more important. This especially holds for application areas like vision, pattern recognition and database mining. In order to fulfill these requirements, throughout the last decade a large number of high performance execution platforms has been proposed for implementing and simulating neural networks. Parallel processor systems have particularly been of interest because they are liable to provide a proper execution platform for exploiting the intrinsic parallelism present in neural networks. Several levels of parallelism can be distinguished, and for each level some execution platforms may be more qualified than others. In this thesis the focus will be on message passing multi-processor systems, and in particular on multi-transputer systems.

The transputer is a microprocessor with processing units, control logic, private local memory and four communication links on one single VLSI device [65, 67]. The transputer can be used in a single processor fashion, but also in multi-processor configurations for building a high performance parallel execution platform. In such a *transputer network*, it is required for the programmer to carefully determine how an application has to be decomposed over the available processors, and how the processors have to be connected with their four links. The two processor topologies used for the artificial neural network simulations in this thesis are a *grid* and *tree*:



(a) *grid*



(b) *tree*

Figure 2.4: *A multi-transputer tree and grid topology. A transputer is depicted as a square box with four lines representing the links in the north, east, south and west direction.*

In the literature, a wide variety of execution platforms have been reported for implementing and simulating neural networks. These range from sequential personal computers, workstations, powerful supercomputers, SIMD and MIMD parallel processors like the connection machine and transputer arrays, to special purpose neuro-hardware. Large scale SIMD parallel processors have been used like the IBM GF11 for implementing backpropagation [124] and the DAP for amongst others Hopfield networks [36]. Obermayer *et al.* [74] describe the parallel implementation of Kohonen self-organizing feature maps, comparing the performance on a transputer array and the Connection Machine CM-2. The latter platform has also been used by Levin for simulating Hopfield-like networks [64] and Singer who achieved 1.3 giga-interconnects per second for backpropagation networks [94]. MIMD parallel processors have been used like the Intel iPSC hypercube for experiments with backpropagation networks [29]. Feldman *et al.* describe the parallel implementation of their Rochester Connectionist Simulator on the BBN Butterfly [34] and transputer arrays are used for a whole range of neural network simulations in [16, 20, 81, 82, 100, 110, 120]. Finally, a lot of results have been published concerning the design and implementation of general or special purpose neural hardware. For example, this was done by using digital signal processors [28, 68] or designing VLSI implementations [27, 43].

In chapter 3, an overview over different parallel processor systems is given, and the transputer architecture is discussed in detail.

2.3 Suitability of MIMD processor systems

When discussing the suitability of an execution platform for neural networks several questions arise. The first is which platform is qualified the best for this application area? In order to answer this question, different platforms have somehow to be compared to each other. This can be done via performance benchmarking, where one or more benchmarks are run on the different platforms and the resulting performance measures are compared. *Performance* is defined as the number of operations related to the problem domain that are computed in one time step. The execution platform with the best performance would thus be the most suitable. However, a further question that arises is would this execution platform still be the best if the size of the target architecture or problem domain changes? When adding more processing, communication or memory resources, the platform is considered suitable if a proportional reduction in execution time (*speedup*) is achieved. The third question is closely related to the former one and concerns the *scalability* issue. The platform and application is scalable if by increasing the processing and memory resources, a comparably larger problem can be solved in the same execution time. The final question that is considered here is what *efficiencies* are achieved when using a certain amount of processors and communication resources. The efficiency for a given speedup is the speedup divided by the number of processors, thus indicating what fraction of the available resources is used productively without waste. In the subsequent chapters, the implementations of parallel neural network simulations (*PNNS*) will all be evaluated from the perspectives of performance, speedup, scalability and efficiency.

The performance of an execution platform for neural network simulations is often expressed in the number of (million) connection updates per second, or *(M)CUPS*. Apparently, MCUPS is a relative term which depends on a combination of a large number of factors, such as the neural network algorithm, its implementation, its size, the computer architecture, available memory, processing and communication resources, etc. This involves that when comparing MCUPS on different execution platforms, all these parameters have to be taken into account. Otherwise, MCUPS would be a **m**eaningless, **c**ompletely **u**seless **p**erformance **s**cale, in analogy with MIPS and MFLOPS as discussed by Dongarra and Gentzsch [25].

2.3.1 Means of estimating performance

There are two ways of estimating the performance of an execution platform for an application, which are described in more detail in chapter 4: performance benchmarking and performance modeling.

Performance benchmarks comprise one or more complete programs or program parts representing one or more classes of applications. By timing a benchmark on an execution platform, its performance for the class of applications can be determined. In Chapter 4 it will be made clear that the use of this method for predicting the performance for parallel neural network simulations has only limited potential. First, there exist no benchmarks for all neural network paradigms. Benchmark results for different applications cannot be compared very easily. Second, a large number of factors determine the performance such as hardware and software environment, the application, its size and the way it is implemented. This involves that a very careful examination of both the benchmark and the application, and the conditions under which the benchmark is run, is required. Furthermore, for answering question regarding speedup and scalability, this method cannot be used.

Performance modeling uses an analysis of the complexity of the execution platform and application in terms of, e.g., memory and communication requirements, arithmetic operations and size of the application. Using the machine specifications comprising individual timings for arithmetic operations, memory accesses and communication, an overall performance measure can be determined. However, again a large number of factors determine the arithmetic performance of a machine and therefore, the resulting performance estimate will not be very precise. Furthermore, the complexity analysis and corresponding evaluation of the factors that determine the performance is an intricate matter, that has to be carried out for each new execution platform and application.

2.3.2 A new, combined approach

In this thesis, a method is presented that combines performance benchmarking and modeling. To estimate the performance of a MIMD platform for a neural network application, some calculation and communication benchmarks have to be measured. Using a model of the communication and calculation costs, the overall time can then be determined. In general, the overall time for training or recall of a neural network application of n neurons, w connections and using p patterns running on a platform of P processors is modeled as the sum of the calculation and communication times:

$$T(P,n,w,p) = T_{calc}(P,n,w,p) + T_{comm}(P,n,w,p) \tag{2.2}$$

The calculation time T_{calc} can be modeled as consisting of the times required for executing a small number of *kernel* functions (e.g., compute a weight change, update a connection, update a neuron). These are executed a large number of times (e.g., per connection or per neuron):

$$T_{calc}(P,n,w,p) \quad = \quad \frac{1}{P} \cdot \sum_i N_i(n,w,p) \cdot t_i \tag{2.3}$$

In (2.3), t_i is the execution time for computing function i and $N_i(n,w,p)$ denotes the number of times the function is computed. Note that a perfect load balance can be assumed, as either n,w or p are large compared to P.

The communication time is modeled as the number of times a single information unit (e.g., a connection or activation value) must be sent over a physical communication link

$(C(P, n, w, p))$, multiplied by the time required for communicating one value:

$$T_{comm}(P, n, w, p) = C(P, n, w, p) \cdot t_{comm} \tag{2.4}$$

If required by the hardware resources or communication patterns of the neural network, a more complicated model for T_{comm} can be used, e.g., by taking setup times into account. This communication model suffices for transputer networks, but a more elaborate model may be required for, e.g., a parallel system containing workstations in a local area network. The needed benchmarks in this method are restricted to measuring the execution time of a small number of kernel functions on one processor and the time needed to communicate a single information unit between two processors. This approach can be classified as kernel benchmarking [5, 46].

2.3.3 Examining the suitability of execution platforms

By predicting the performance for a different number of processors P_1 and P_2, the suitability of the platform in terms of speedup, efficiency and scalability can be examined. The plain performance number can be used to find out how close the peak performance of the platform can be reached. This gives some qualitative statements about the efficiency of the implementation and use of the available resources. The speedup for P processors is defined as the time it requires to compute a problem on 1 processor divided by the time it takes to compute it on P processors (Eq. 2.5). The parallel speedup for P_1 and P_2 processors can be defined as (Eq. 2.6):

$$S(P, n, w, p) = \frac{T(1, n, w, p)}{T(P, n, w, p)} \tag{2.5}$$

$$S^{par}(P_1, P_2, n, w, p) = \frac{T(P_1, n, w, p)}{T(P_2, n, w, p)} \tag{2.6}$$

If the speedup shows a linear characteristic, the target platform can be used efficiently for the problem. If the speedup only increases slowly with the number of processors, or if it even drops, this usually means that the communication overheads become too severe. This can be translated in the statement that the available communication resources are insufficient for the problem or that the communication/computation ratio is too high. The latter means that the amount of work each processor has to do is too low compared to the required communications. It will be shown in the subsequent chapters that indeed, when increasing the problem size, the achieved speedups increase too.

The speedup as defined above is also known as fixed-size speedup. The problem size is kept constant and the number of processors is increased. Following Amdahl's law [1], there is always an upper limit $1/s$ on the speedup, where s represents the sequential part of the parallel algorithm. If, as is often the case in simulation studies, researchers want to increase their problem size (e.g., grid size in weather forecasting or image processing, or the number of patterns or connections in artificial neural networks), they want to know whether their application is scalable on a specific machine architecture. When examining the scalability of a parallel program, the problem size is not kept fixed but instead, it is increased proportionally with the amount of processors. A parallel problem is called scalable if when increasing both the problem size and the number of processors, the total execution time of the parallel program stays constant. This only holds if what we call the scalability factor equals one (2.7). The problem size depends on the number of neurons and connections in the neural network and on the number of patterns. This is denoted as (n, w, p):

$$f^{scal}(k, P, (n, w, p)) \quad = \quad \frac{T(P, n, w, p)}{T(k \cdot P, k \cdot (n, w, p))} \tag{2.7}$$

In [42], the so called scaled speedup is introduced in order to overcome the problems imposed by Amdahl's law. According to Gustafson, "speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size". In [35], the definition of scaled speedup is given, which exactly matches our definition of the scalability factor times k:

$$S^{scal}(k, P, (n, w, p)) \quad = \quad k \cdot f^{scal}(k, P, (n, w, p)) \tag{2.8}$$

In the sequel, the term scalability will be used meaning scaled speedup and the term speedup will indicate fixed-size speedup. In general, scalability will result in higher numbers than speedup. Therefore, using speedups to examine the suitability of an execution platform for an application could give the indication that the platform is not well suited, whereas using scalability would indicate a better suitability of the platform. In this thesis, both scalability and (fixed-size) speedup are considered.

The method presented here will be validated quantitatively for different decomposition methods and processor topologies (e.g., grid, tree) in chapters 6 and 7. Depending on the exploited decomposition techniques and processor topologies, the modeled T_{calc} and T_{comm} may differ strongly. Therefore, the method requires that for each topology and application used, an analysis has to be made of the calculation and communication times. Using the analysis and the times associated with the measured kernel benchmarks, the method can be used to predict the performance of any MIMD multi-processor system. The method is a scalable performance prediction method, because based on the times measured on one processor and one communication link, predictions can be made for larger processor systems. Furthermore, for each neural network application, the analysis of its calculation times and the identification of the corresponding kernel functions has to be performed only once. Its calculation times for new target execution platforms can subsequently be estimated by measuring its kernels. This also holds for the communication times on a specific execution platform. Once its communication times are estimated, they can be used to predict the performance for any neural network application. Therefore, the method is called a scalable, general purpose performance prediction method.

This combined method will be discussed in chapter 4. It is targeted on MIMD parallel processor systems that communicate with each other through message passing. An overview over MIMD parallel processor systems is given in chapter 3. Chapter 4 covers the design, implementation and performance evaluation of a message passing communication layer tailored for distributed neural network simulations. In chapters 5 and 6, the performance prediction model will be applied on two popular neural network models, the multi-layered perceptron (backpropagation) network [83] and the Kohonen self-organizing feature map [58]. Three multi-transputer systems are used for the quantitative validation of the models. The system located in Nijmegen contains 64 T805 transputers. Two other systems were made available by the University of Amsterdam for experimentation. The GCEL-512 containing 512 T805 nodes and the PowerXPlorer containing 32 PowerPCs allowed for the examination of the performance, speedup, scalability and efficiency issues for a larger number of processors and for processor nodes with higher computing power.

2.4 Neurocomputing and Neurosimulators

Neural networks are being used in many commercial and research applications. People using or developing artificial neural networks are involved in the *world of neurocomputing*. Despite of the growing popularity of neural networks, there have always been a number of oppositions against using them. Researchers in the theoretical field are not pleased with the fact that they do not really know what is going on 'inside' the network. What do values of parameters like "weights" and "biases" mean when they are tuned for an application. Potential neural network users working in control or industry are complaining about the reliability and precision of neural networks. If hard real-time responses are required and a neural network hesitates or gives an inaccurate reaction, disastrous failures may occur. Finally, the long training times required for tuning a network are a nuisance and may justify the use of high-performance target hardware like the transputer systems described in this thesis.

In order to a) examine the inner workings of a neural network, b) to increase its generalization precision through experimentation, c) to support the tuning of its parameters, or d) to provide fast execution platforms, powerful tools are required. Such tools, which assist a user in the world of neurocomputing are called neurosimulators. We define the term *neurosimulator* as any set of software and/or hardware components dedicated to designing, controlling, monitoring, manipulating, or (fast) executing neural network simulations. In the FAQ of `comp.ai.neural-nets`, an elaborate list of neurosimulators is available. Most neurosimulators have emerged from the needs for support during experimentation. Existing neurosimulators like the RCS [40], Aspirin-Migraines [63], NNSIM [106], MetaNet [70] or Genesis [123] are all built by research teams who were already doing neural network research before starting to build their tools. As the number of people involved in using neural networks has increased, a potential market for neurosimulators has come up. This has led to the situation in which not only traditional neural network researchers, but also electrical or chemical engineers, computer scientists, and all kinds of other academic or commercial institutes, are confronted with the problems of neural network design. Commercially available neurosimulators are for example Mimenice [86], NeuralWorks [72] and BrainMaker [97]. Pygmalion [2], SNNS [30] and PREENS [112] are examples of neurosimulators built by computer scientists. In this thesis, a new approach to neurosimulators is given. The justification, design and implementation of what we call an action-oriented neurosimulator is presented.

In this introduction, a taxonomy of the users involved in neurocomputing is given. Based on the different ways they work with neural networks, and on the typical actions that are carried out when doing neurocomputing, a conceptual model of the *neurocomputing life cycle* will be introduced. By examining the neurocomputing life cycle, the requirements can be identified concerning *which* actions have to be supported by a neurosimulator. In Chapter 8, the *neurocomputing environment* will be discussed. Also, an overview over existing neurosimulators is given. By examining neurocomputing environments, the requirements on *how* the typical actions have to be supported by a neurosimulator can be distinguished.

2.4.1 Users in the world of neurocomputing

Four classes or groups of users associated with neural networks can be identified: 1) model builders, 2) tool builders, 3) applied researchers, and 4) end users.

◇ *Model builders* are people involved in basic or applied neural network research. They can be found in research departments like cognitive science or biophysics. Their goal is to obtain insight in the functioning of (parts of) the biological brain, or to build artificial neural networks that are suited for a specific application range. Though in many occasions the models they come up with are not simulated on a computer because they merely exist in mathematical formulas, computer simulations are often used to validate the behavior of the model. Particularly of interest is that in such cases these users tend to not use neurosimulators for implementing and simulating their neural network model, but rather program it from scratch in a language like C [56]. The main reasons for this are that most neurosimulators run slowly compared to tailor made programs, and furthermore that model builders are not willing to comply to the requirements of the neurosimulator for adding or changing a (new) neural network model.

◇ *Tool builders* are people that implement neurosimulators. Their interest in building them can originate from a possible need that came up during experimentation. Therefore, tool builders can very well be people that belong to classes 1 and 3. Furthermore, tool builders can have as a goal to do research in building neurosimulators or to come up with commercially interesting packages.

◇ *Applied researchers* can belong to any group of people who are using neural networks for some application. Typically, the more one is involved in training a neural network and tweaking its parameters and architecture, the more one is willing to try out tools and/or neurosimulators. Eventually, this may lead to building a neurosimulator which is fine-tuned for the application. As a final stage of tuning the application, typically the neural network part of it is integrated in the ultimate product.

◇ *End users* are people who use such an end product. The product is tailor-made for the application the end user is interested in. The final end product may be the result of a combined effort of the end users, model builders, tool builders and applied researchers. Normally, neurosimulators are not used by this group for experimentation but for production purposes only, as the end user has very limited experience with neural networks and merely wants to use them to solve his particular problems.

People belonging to the model builders or applied researchers impose the highest requirements on neurosimulators, especially toward extendibility and flexibility concerning new models, new pattern formats and new (graphical) tools. They want to be able to implement and validate new models, to change existing models or to combine several models into a new, heterogeneous architecture.

2.4.2 The neurocomputing life cycle

The neurocomputing life cycle is considered as a multiple feedback loop over several stages. In each stage a certain action is performed and one can jump from each stage to each other. The stages in the neurocomputing life cycle can be grouped into four categories: *initiation*, *tuning*, *testing* and *production* (see Figure 2.5):

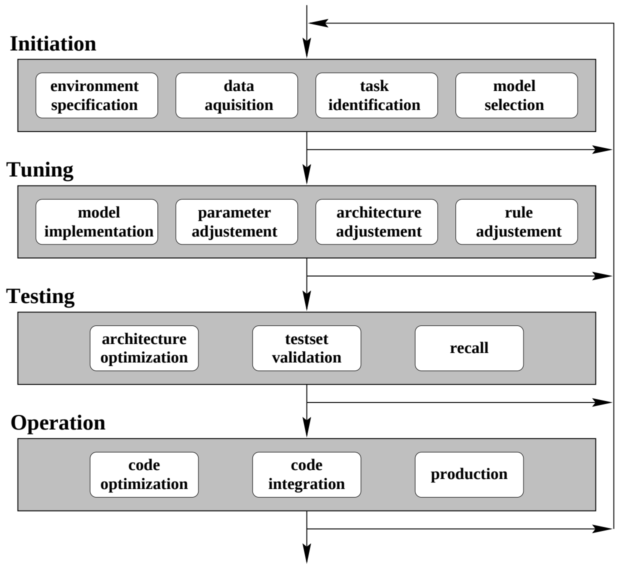


Figure 2.5: *The neurocomputing life cycle.*

Initiation

In the initiation phase it is decided what task has to be performed. This can be the development of a new neural network architecture (typically done by model builders), or any application in which a suitable neural network has to be found, tuned for a certain dataset and integrated within a certain environment. Stages in the initiation phase are

- *Task identification*, which reflects the aim of using neural networks.
- *Environment specification*, which states the requirements concerning input and output streams, execution speed and accuracy of the neural network to be used.
- *Model selection*, which depends heavily on the task to be performed.
- *Data acquisition*, which is partly ruled by the environment and has constraints imposed by the neural network.

Note that the neural network or the application may require that the dataset is formatted or preprocessed in a specific manner. In the initiation phase it is decided how to integrate environment, neural network and dataset.

Tuning

After the initiation phase, tuning may involve finding the optimal neural network architecture, parameter settings and activation and training rules for a given dataset. If the selected task cannot be accomplished within certain limitations, it may even be so that the conclusion of the tuning phase is that a different task has to be selected, or that the task cannot be accomplished. Stages in the tuning phase are

- *Model implementation.* At the initiation phase, the neural network model to be used is selected. If a new model has to be developed, or if the chosen model is not supported by available neurosimulators, implementation of the model is required.
- *Architecture adjustment*, changing number of neurons, weights, thresholds, etc.
- *Parameter adjustment*, changing learning rate, weight values, thresholds, etc.
- *Rule adjustment*, changing activation or learning mechanisms.

At any moment during tuning, a different neural network architecture or model may be selected, or a different dataset may be constructed. This means that a feedback jump is made to the initiation phase.

Testing

Once a neural network is tuned for an application, the testing phase is carried out. For a different dataset than the one the network was tuned with, its generalization capabilities are tested. The test data will have to represent sufficient relevant samples of the eventual application that is executed in the operational phase. In order to arrive at an optimal network architecture, often many feedback loops between tuning and testing phases are performed. Stages in the testing phase are

- *Recall*, which is used to compute the network generated output for a given input pattern.

- *Architecture optimization*, for example a technique called pruning can be used to find a minimum number of hidden layers and nodes for which a backpropagation neural network still is capable to perform well on a test set.

- *Test set validation*, which examines the performance of the neural network for the test data. The performance may represent execution speed, accuracy or reliability.

Also after the test phase it may be concluded that the neural network is not able to perform well enough within the given environment, for the given dataset, network architecture, and tuned parameters. This conclusion may lead to feedback jumps to the previous phases.

Operation

Once the neural network architecture and parameters are tuned and tested, the network is ready for the eventual operational phase. Further optimizations are possible (e.g., extracting only relevant program parts from the neural network simulation program, optimizing the code, or implementing the network in hardware), but in general no changes are made to the architecture and parameters itself. Especially if the neural network is used as one step in a range of processing stages, in many cases the neural network code is integrated in the eventual application. The stages in the operational phase are

- *Code optimization*, for example using in-line assembler primitives, loop unrolling and other techniques or by using hardware accelerators.

- *Code integration*. The full end-application may already have the neural network integrated in its environment. However, especially when using neurosimulators to tune and test the network, this stage may involve extracting a stand-alone program and integrate it within the environment.

- *Production*, is the final stage during which the network is used to accomplish the overall task.

2.4.3 An engineering approach to neurocomputing

Another approach to this life cycle is given by Whittington and Spracklen in [119] . Their paper is one of a very limited number of publications which discuss the neurocomputing life cycle. It is an engineering approach, in which six phases are distinguished in the development of a neural network application: assessment, specification, design, implementation, evaluation and delivery. In the *assessment* phase, the aim is to find out whether neural networks are a feasible or appropriate technology for the task to perform. Limitations of neural networks like the ones mentioned at the beginning of this chapter have to be considered. The *specification* phase has to consider several issues, which are the nature and type of the application, the I/O representation of the data, and its availability. The nature of the application can be either stand-alone or embedded, where the latter means that in later phases the neural network has to be coupled to other components of a larger system. The I/O representation is specified by the system, where the representation required by the neural network is only specified during the design states. The availability of the data is of importance during the specification phase because in many occasions, data acquisition may be a time or money consuming process. At this moment during the life cycle the decision may be made to choose e.g., a self-organizing neural network because of the unavailability of supervised data. In the *design* phase, the neural network model, possible pre- and post-processing of the data, the runtime requirements and I/O requirements of the neural network are identified. After the specification and design, the neural network is implemented during the *implementation* phase. In software engineering it is commonly assumed that once specification and design are correctly specified and validated, the implementation is a straight forward issue. If the implemented system stands the testing and evaluation tests, it is ready for acceptance. However Whittington and Spracklen mention that the key issue in the neurocomputing implementation phase is the validation of the implementation for its correctness, as often implementation errors will not be observed because of the adaptive nature of neural systems. Because of this feature, a neural network will still continue its operation despite of it being incorrectly implemented. Indeed, many discussions in the news-net pages of `comp.ai.neural-nets` indicate that users have incorrectly implemented their neural network and wonder why it produces strange or partly correct results. The *evaluation* phase is carried out as depicted in Figure 2.6. The final phase is the *delivery* phase during which the implemented system is installed within its environment.

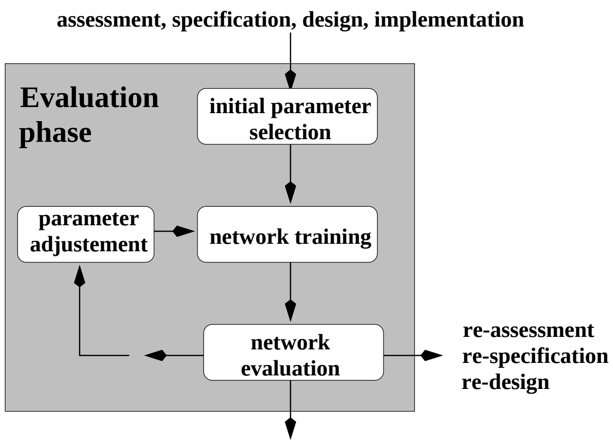


Figure 2.6: *Evaluation phase in the methodology of Whittington and Spracklen. This figure is drawn from Figure 3 in [119]. The evaluation phase corresponds with the tuning and testing phases in Figure 2.5.*

When comparing the neurocomputing life cycle as depicted in Figure 2.5 and the development cycle of Whittington and Spracklen, many similarities can be found. Note that in Figure 2.5 fourteen different stages are identified. To compare these to the six phases of Whittington and Spracklen with the initiation, consider Table 2.1:

phases	stages in Figure 2.5
assessment	task identification
specification	data acquisition, environment specification
design	model selection, environment specification
implementation	model implementation
evaluation	the stages associated with tuning and testing
delivery	code integration, production

Table 2.1: *Similarity between stages listed in Figure 2.5 and the methodology of Whittington and Spracklen.*

The assessment, specification and design phases of Whittington and Spracklen are performed in the initiation phase. The implementation and some of the evaluation phase are contained in the tuning phase, whereas some of the evaluation phase also occurs in the tuning phase. Because the development cycle of Whittington and Spracklen is specifically targeted at the use of neural networks to perform a specific task within an application, their most important phase is the implementation phase. This is emphasized in [119]. However, depending on what the user of neural networks wants to achieve, in many cases the implementation phase is completely superfluous, especially if the selected neural network model (in the initiation phase) is already supported by an available neurosimulator.

Considering the four groups of potential neural network users and the neurocomputing life cycle discussed in this section, the respective important phases are listed in table 2.2:

user class	interest
model builders	model selection, model implementation
tool builders	model implementation
applied researchers	task identification, data acquisition, environment specification, tuning, testing and operation
end users	operation

Table 2.2: *Phases and stages depending on user interests.*

2.4.4 Towards an action-oriented neurosimulator.

In Chapter 8, several existing neurosimulators are discussed, like Pygmalion [2], the Rochester Connectionist Simulator [40], Aspirin/MIGRAINES [63] and Genesis [122]. They are considered from two perspectives, the perspective of the user and its environment, and the perspective of the neurosimulator and its environment. Neurosimulators can be distinguished in three categories, application-oriented systems, algorithm-oriented systems and general programming systems. For example in [77], [80], [109] and [116] the characteristic features that neurosimulators offer are listed. Neurosimulators can have a graphical user-interface, an algorithm library, and support for building new models. They can contain visualization and monitoring tools, neural network description languages, application specific tools, and dedicated hardware accelerators. Based on the examination of the environment in which neurosimulators are used, potential user-requirements, and the pros and cons of features that existing neurosimulators contain, in this thesis a new neurosimulator is proposed. It is called PREENS, *a parallel research execution environment for neural systems*. The requirements that PREENS has to fulfill are:

1. Provide a general purpose user-interface suited for the monitoring, visualization and control of any running neural network simulation program.

2. Provide an interface definition via which new or existing simulations can be coupled to the user-interface without much effort.

3. Provide a communication and control interface via which both distributed and sequential simulation programs can be controlled.

Three components of PREENS were developed to make sure to fulfill these requirements. The first is a small specification language for specifying action-oriented program descriptions. The second is a set of interface definitions that use an action-oriented program description to access pieces of neural network simulation data and code. The third is a graphical user-interface called CONVIS via which programs and associated tools can be controlled.

2.4.5 The program description

The major insight that led to a system meeting these requirements was that most neural network simulations only implement a limited number of *actions*, being the loading and saving of neural network data or stimuli, the initiation and control of a training or recall session, and the final operation of the tuned simulation program. By specifying this limited set of actions, and providing an interface that can use the specification for control, visualization and operation of the program, the requirements listed above can be met. This involves that whereas most other neurosimulators are based on a general description of the *neural network*, PREENS is based on a description of the neural network *program*. This *program description* uses what I call an action-oriented model.

In such a model, each program can be described by specifying the actions it implements. Each action can have a number of objects associated with it, e.g., parameters, variables, data, options and settings. Figure 2.7 depicts the conceptual model of a PREENS action:
A *program description* contains specifications of a number of such actions. Each action has parameters, options and settings for installing its initial configuration. And it has variables and data whose values can be changing during execution of the action. The action-oriented model and program description are explained in Chapter 9.

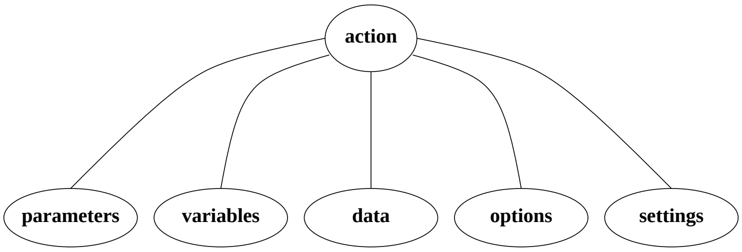


Figure 2.7: *Conceptual model of the* PREENS *action.*

2.4.6 CONVIS , the user-interface for control and visualization

CONVIS [116] is the general purpose user-interface that manages a running neural network simulation program and any number of associated tools. It provides an environment which takes care of the control of actions while making minimal assumptions about the way in which they are implemented. It uses a set of interface definitions using a program description, for data exchange with and control of running neural network simulation programs and associated tools. The environment of PREENS is discussed in chapter 9, and consists of the manager CONVIS and a set of neural network programs and tools (see Figure 2.8).

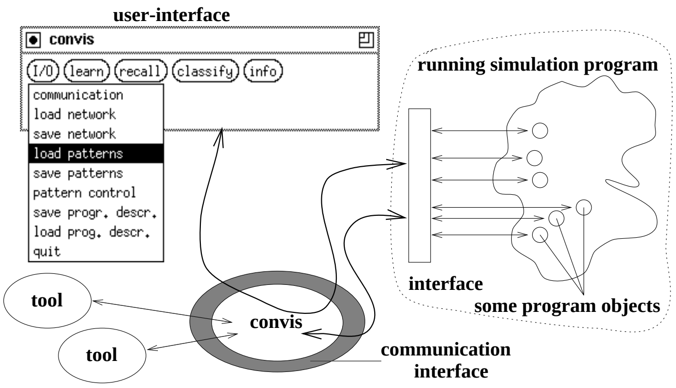


Figure 2.8: *The* PREENS *neurosimulator environment.*

2.4.7 Applications of PREENS

Each of the neural network simulation programs contained in the PREENS algorithm library is tested on a number of data sets. For example sets containing echographic liver data and sets containing printed characters were examined. The most important application considered is the classification of ground cover classes using remotely sensed imagery. In collaboration with Ron Schoenmakers of the Joint Research Centre at Ispra, Italy, this work resulted in a paper for the IGARSS '95 conference held in Fiorenze, Italy [87]. This application will be used as a running example in the last chapters of this thesis.

3

Parallelism and the transputer

Outline

This chapter briefly describes the parallelism that can be observed in computer systems. The class of MIMD parallel processors to which transputer systems belong is discussed in more detail. The architecture of the T8xx transputer is presented and the differences with its successor, the T9000, are listed. Three transputer systems are described, the NSC containing 64 T800 processors, the GCEL containing 512 T805 processors, and the PX, a system containing 32 PowerPCs. Furthermore, the most significant features of how to use the operating systems Helios and Parix are described.

3.1 Parallelism in computer systems

The overall goal of exploiting parallelism is to finish a job in a smaller amount of time, or to handle more (distinct) jobs in the same amount of time. This goal is achieved in current computer systems by exploiting parallelism on several levels of detail. These can be classified in increasing level as: (1) bit level, (2) instruction level, (3) function unit level, (4) function level, (5) CPU level, (6) data level, (7) program level and (8) computer level.

On the lowest level – *the bit level* – the bandwidth of the data bus (e.g., 8,16,32, or 64 bit wide) determines how many bits of information can be transferred between processor units in parallel within one clock tick. Similarly, the ALU or FPU can be of, e.g., 16, 32 or 64 point which means that that many bits of information are processed within one clock tick. I/O can be performed via parallel ports, transferring a number of bits in parallel. On an even finer level of detail, the smallest components of a computer systems all work in parallel. Current technology of making VLSI or WSI implementations allows for a very high number of components packed on a small area of silicon to operate simultaneously. For example, several special purpose hardware computers have been designed that implement parallel neural networks in neuro-asics (application specific integrated circuits).

On *the instruction level*, parallelism is exploited by parallelizing the instruction cycle. A well known method is instruction pipelining which executes instruction fetch, instruction decode, operand fetch, execution and store of results. A series of operations can be computed significantly faster in this way. An example instruction pipeline is depicted in Figure 3.1.

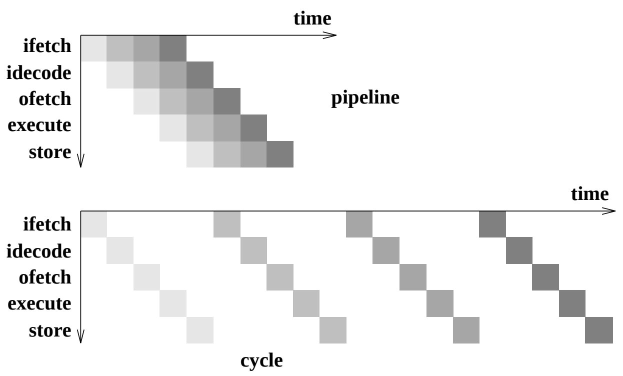


Figure 3.1: *Instruction pipelining versus normal instruction cycle, time in clock ticks.*

On *the function unit level*, parallelism exists in the occurrence of multiple functional units, e.g., one or more ALU, FPU. Furthermore, for example the FPU can be even more subdivided in, e.g., floating point adder/subtracter and floating point multiplier/divider. By simply replicating functional units and using pipelining, multiple arithmetic operations can be performed in parallel or simultaneously. One of the key issues on efficient pipelining is how to provide the pipeline with data (instructions,operands). The memory bandwidth (the average number of information units that is accessed per time unit) and the instruction and data access characteristics highly determine the performance of a pipeline processor.

The *function level* can be characterized by the existence of several dedicated hardware components that each perform a specific task, like disk I/O, inter-processor I/O (communication), memory interfaces and arithmetic units.

More and more often, current computer systems facilitate several CPU (e.g., Sun ultra, IBM RS6000 cluster). This results in parallelism on the *CPU level*. The CPUs share multiple resources like shared memory, disk drives and communication hardware and therefore the performance of such computer systems is highly dependent on the managing operating system. The operating system decides whether shared resources may be accessed in parallel or not. Applications that require many non-shared system calls (e.g. file I/O) may suffer enormously from contention in resource accesses, whereas applications that exhibit much computational efforts may run highly efficient.

Data level parallelism is observed in computer systems that exhibit the possibility to compute many (simple) operations on individual data elements in parallel. These computer systems are known as SIMD (single instruction multiple data) parallel processors. Array processors like the massively parallel processor (MPP), and multi-processor systems like the Connection Machine are SIMD processors. These systems feature a large number of relatively small processing elements and an interconnection communication network for routing of data and instructions. Each processing element executes the same set of instructions in a lock-step fashion, where the instructions are broadcast by a centralized control unit. A typical SIMD parallel processor architecture is depicted in Figure 3.2.

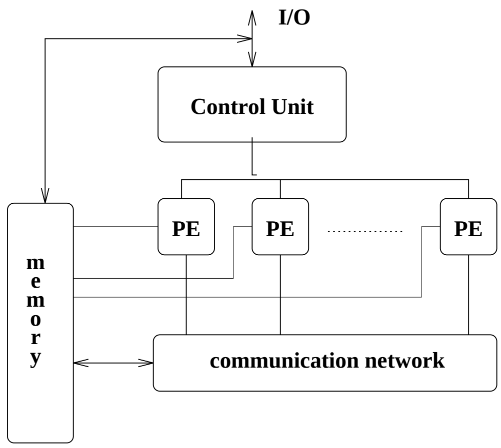


Figure 3.2: *A general SIMD parallel processor architecture.*

Another class of computer systems that show data parallelism are vector supercomputers like the Cray-1, CDC STAR 100 and Cyber-205. Besides providing scalar operations, such systems are designed to operate on data arranged in a regular, homogeneous structure like in matrix or vector formats. Vector operations are performed in a pipelined fashion on subsequent vector elements (see Figure 3.3) and there may exist multiple vector pipelines.

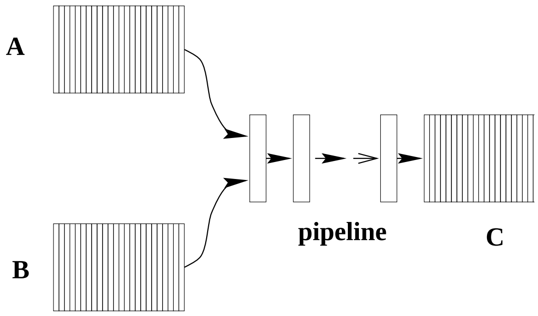


Figure 3.3: *Simplified depiction of a pipelined vector computation.*

Furthermore, vector pipelines may be chained thus holding intermediate results ready for execution before storing them back into memory. The concept of pipeline vector processors can be classified as MISD (multiple instruction single data), as different stages in the pipeline perform different operations, but operate on only one scalar or vector data element. Vector processors are characteristic for their very high instruction throughput of hundreds of MFLOPS. A typical vector processor architecture is depicted in Figure 3.4.

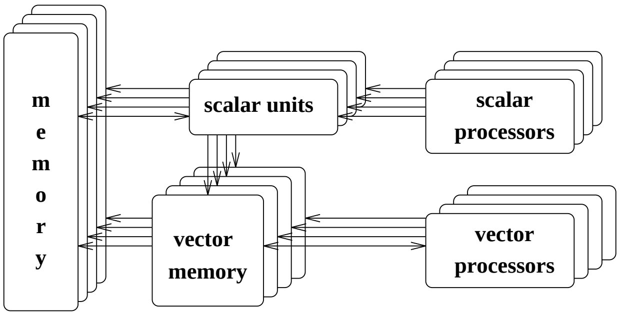


Figure 3.4: *Typical architecture of vector processors.*

Program level parallelism can be observed in multi-tasking and/or multi-user environments, where several independent tasks run via time-sharing on one or more processors. In case of several processors, programs can run in parallel. In case of one processor, it is the task of the operating system to schedule tasks and take care of swapping code to and from memory. Program parallelism is the most commonly used type of parallelism, which is exploited in almost all workstation and personal computer systems available today.

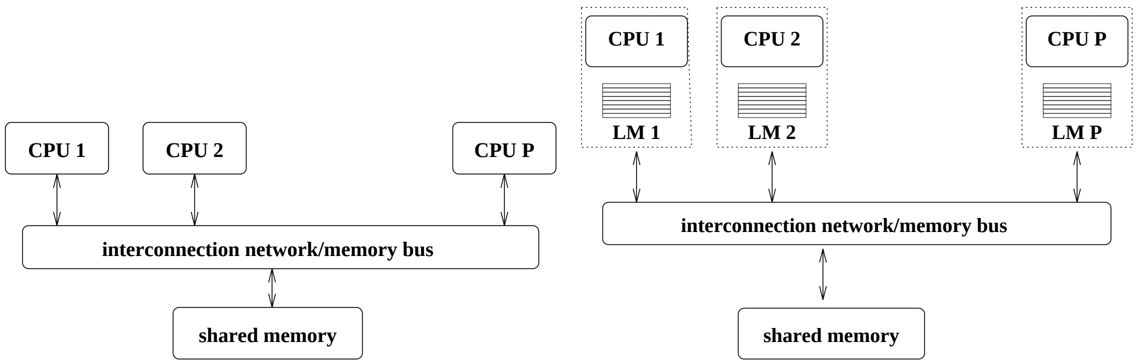
The highest level of parallelism is on the *computer level*. The class of computer systems that consist of several processors, each having its own CPU(s), arithmetic units, I/O components, etc., is called MIMD (multiple instruction multiple data). These systems are considered general purpose, as each processor in the parallel processor network may compute a separate program, different from programs running on other processors (i.e. multiple instructions) on a distinctive set of data (multiple data). The Inmos transputer can be used as a processing element for building MIMD parallel processors. As our target hardware contains transputer systems, in the rest of this chapter the attention is focussed on computer systems that belong to the class of MIMD parallel processors.

3.2 MIMD parallel processor systems

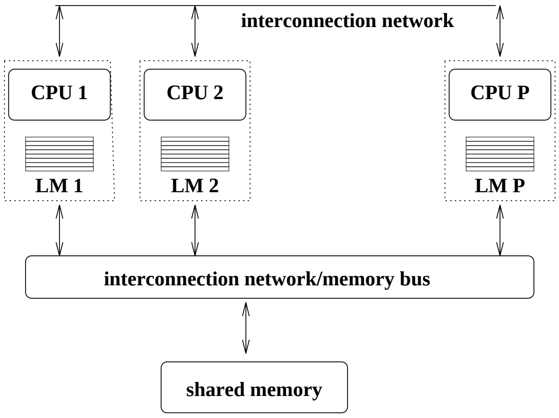
A typical MIMD processor network consists of a set of processing elements and an interconnection network for transferring data between processing elements and external memory. Basically two kinds of MIMD parallel processors exist which are shared memory and distributed memory systems.

3.2.1 Shared memory systems

As any other MIMD architecture, shared memory systems contain a number of computation elements (CPUs). Salient feature is that there exists some means via which different CPUs can map into some global address space (shared memory). This can be done via some common communication path, which can be a memory bus, some (multi-stage) interconnection network or a combination of a bus and interconnection network. The available memory can be completely global (Figure 3.5(a)), each processor can have its own local memory while sharing a global memory pool (Figure 3.5(b)), or processors can somehow share certain parts of each processors local memory (Figure 3.5(c)). Also some combinations of these architectures are possible.



(a) *Shared memory without local memory.* (b) *Shared memory with local memory (LM).*



(c) *Shared local memory.*

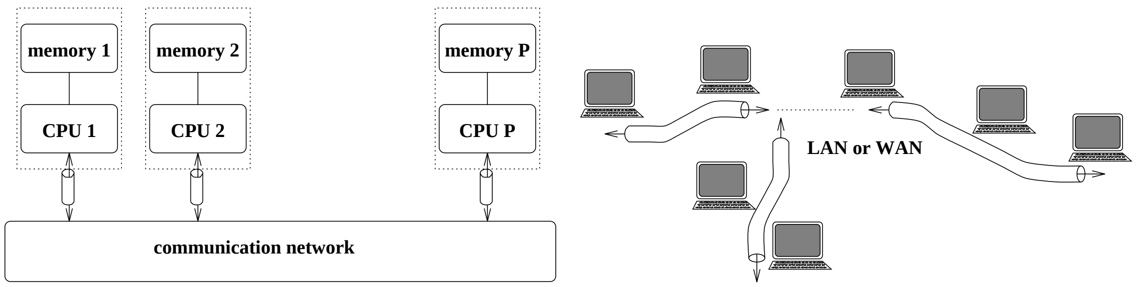
Figure 3.5: *Shared memory processor architectures.*

Bus based shared memory systems use for example standard buses like the VME bus, the NuBus and Multibus. Each processor requests for read/write operations via a bus controller. In order to prevent two processors from writing in the same memory location, a means of determining the availability of the bus is required. Several algorithms exist for scheduling the bus access over the processors, such as daisy chaining, round robin, fifo, lifo and polling. Essential in these systems is the ratio of local cache memory and global memory. If the number of processors is large compared to the bandwidth of the bus, shared memory systems suffer from bus contention, resulting in severe performance degradations. One solution to bus contention is the use of hierarchical multiple buses, resulting in a network of bus communication layers. Whereas a single bus avoids shared memory systems from being scalable, using a multi bus design allows for scalability.

Shared memory systems with local memory per CPU are characterized by a interconnection network, a virtual or real shared memory part, a global shared memory part and a number of processing elements that contain private local memory. The interconnection network (typically a network based on crossbar switches) allows for individual processing elements to access local memories from other processing elements and the global memory. A crossbar switch uses several components to provide all-to-all communications between a number of processing elements. By combining crossbar switches, multi stage interconnection networks can be built to couple clusters of processing elements, resulting in scalable parallel processor architectures. Example of a system with shared and local memory is the BBN butterfly.

3.2.2 Distributed memory MIMD systems

The typical architecture of a distributed memory system consists of a number of processing elements connected via some communication medium. Each processing element can be considered as a stand-alone computer which has its own CPU, memory and communication ports (see Figure 3.6(a)). The Intel IPSC/1 and IPSC/2, the NCube and transputer systems are examples of parallel processor networks with a number of processor elements, each representing a stand-alone computer. The processing elements are connected via a communication network and are equipped with one or more communication processors or link interfaces. The *Intel hypercubes* consist of processing elements with standard 80286 (IPSC/1) or 80386 (IPSC/2) processors. Each processing element contains a direct connect routing module (DSC) which has 8 communication links, one of which is reserved for external I/O. By chaining several DSCs, it is possible to dynamically establish a physical communication path between any two processors. Intel hypercubes have dimensions of 3 (P=8) up to 7 (P=128), delivering a range of peak performances of 4 up to 1280 MFLOPS depending on the availability of scalar or vector co-processors. The *NCube*, developed by the NCube Corporation, covers a range of machines from 16 (NCube/4) processors and 1 I/O system up to 1024 (NCube/10) processors and 8 I/O systems. Each processing element contains a 32-bit processor with local memory and communication links to other processing elements. The I/O systems provide a very high potential bandwidth for transferring code and data between external devices and the hypercube network. Each node has 22 communication channels, 20 of which are used to connect to other processing elements and 2 of which are used for system I/O. *Transputer systems* are extensively discussed in the rest of this chapter.



(a) *Distributed memory.* (b) *Connected via LANs or WANs.*

Figure 3.6: *MIMD parallel processor architectures.*

Remote MIMD systems are essentially stand alone computer systems that communicate via local or wide area networks, such as networks of workstation clusters (see Figure 3.6(b)). Currently, many operating systems designed for parallel computer architectures are ported on these machines. The advantages concerning portability, prototyping, debugging, cross-compilation and heterogeneous network resource utilization are numerous compared to what is currently available on most native operating (or runtime) systems that run on parallel processor systems belonging to the other classes listed above. Parallel operating systems like for example CS Tools, Express, Helios, Parix, and message passing standards like PVM and MPI are successfully used in many high performance applications and it is believed that developing parallel applications on workstation clusters requires less efforts than on native systems.

3.3 The Inmos transputer

The Inmos transputer [65, 67] has processing units, control logic, private local memory, and four communication links on one single VLSI device (see Figure 3.7). Transputers can be used in a single processor fashion for applications like control, but also in multi-processor configurations for building a high performance parallel execution platform for large scale applications.

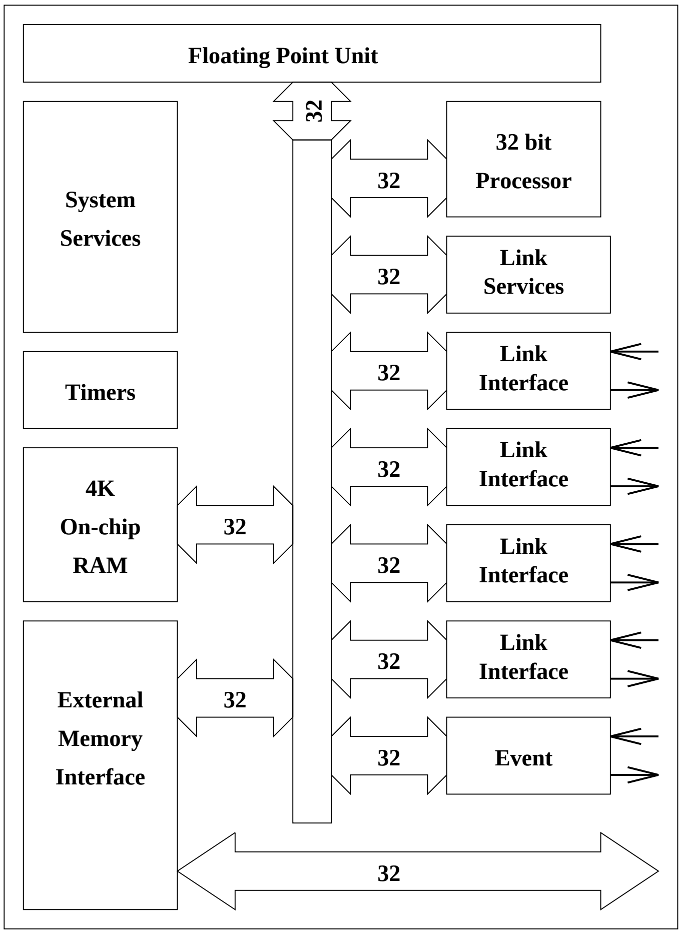


Figure 3.7: *The T800, a 32 bit microprocessor with 64 bit floating point unit. Transputer Reference Manual, Inmos Ltd. [65].*

The T414, T800 and T805 transputers contain several components connected via a 32-bit bus. Via the bus the transputer can access internal memory (2Kb or 4Kb respectively) and using the external memory interface the external RAM (typically 1,4,8,16 Mb) can be accessed. The experiments discussed in this thesis have been performed on three transputer systems, the Nijmegen SuperCluster (NSC), the GCel-512 (GCEL) and the PowerXPlorer (PX). The latter two systems are located at the University of Amsterdam. The NSC consists of 64 T800 transputers with 4 Mb external RAM and a clock speed of 25 MHz each. The transputers in the GCEL are T805 transputers and have the same amount of memory but a clock speed of 30 MHz. The speed of each of the communication links is 20 Mbits/second, delivering a total bandwidth of 80 Mbits/second. The PX is described later in this chapter.

A transputer can address 4 Gb of consecutive memory, part of which contains the on-chip RAM. The CPU is a RISC processor using only six registers for sequential processing. Three registers (A,B,C) are contained in an evaluation stack used for integer and address arithmetic. Three more registers are the workspace pointer which points to a memory area where local variables are kept, the instruction pointer which points to the next instruction and the operand register which is used in the formation of instruction operands (see Figure 3.8). The CPU and FPU can operate in parallel, for example the CPU may perform some address calculations while the FPU executes a floating point operation (the FPU has some additional registers).

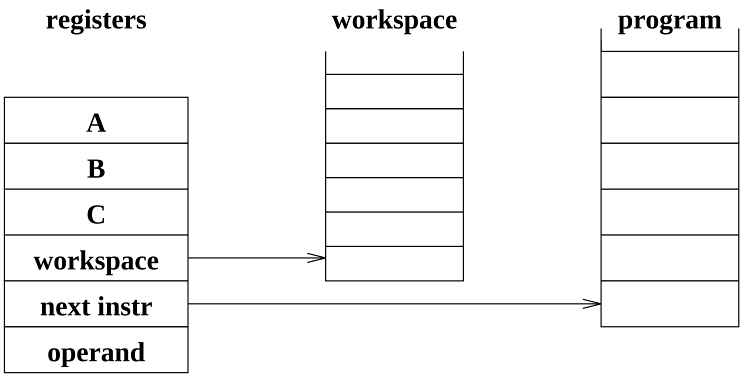


Figure 3.8: *The transputer registers. Transputer Reference Manual, Inmos Ltd. [65].*

The design of the transputer is based on the parallel programming language OCCAM [51], which is an implementation of the language CSP (communication sequential processes) developed by Hoare [47]. In the OCCAM model, an application is described by multiple concurrent processes that communicate via communication channels. Processes can run concurrently on one processor or in parallel on more than one processor. Conceptually, the same OCCAM program can run on one or on several processors. Communication between processes can be done via local (internal) channels implemented through local memory, or via remote (external) channels between neighboring processors. Remote channels are implemented via link interfaces. Both the scheduling of concurrent processes on one processor and the communication via link interfaces is implemented in the transputer microcode.

3.3.1 Process scheduling

The transputer can run multiple processes concurrently at two priority levels (high priority and low priority), where each process has its own set of registers. Time sharing of low priority processes is performed via the micro coded scheduler and only occurs after certain instructions (de-scheduling points) which leave the A,B,C and FPU registers undefined. Because scheduling is performed in hardware and the transputer has only a small number of registers that have to be saved, context switches between processes can be executed extremely fast (in the order of μseconds). Processes running concurrently on one processor can be either active or inactive. Active processes are either being executed or are in a list of processes waiting to be executed. Inactive processes are in a list of processes waiting for some I/O or timer event to occur, and do not consume any computation time.

High priority processes are typically interrupt service routines. They are descheduled only upon termination or when they are in an inactive state. Low priority processes are scheduled only if no high priority process is active. For both priorities a queue of active processes waiting to be executed is maintained. The queue is a linked list of process workspaces, implemented using two registers, one of which points to the first process in the list (front), the other to the last (back). The workspaces contain a process' local variables and some status information. For example consider four processes, P,Q,R and S, running concurrently. The running process is S and the first process to become scheduled is P. When switching S and P, the instruction pointer of S is saved in its local workspace, the workspace and next instruction registers are fetched from P's workspace and the front and back pointers are adjusted (see Figure 3.9).

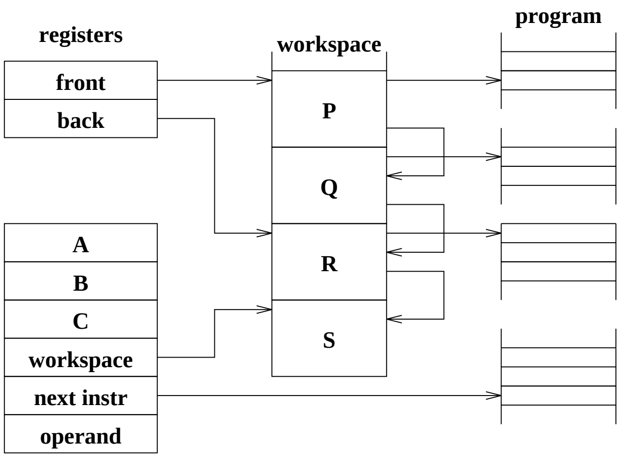


Figure 3.9: *Linked process workspace list. Transputer Reference Manual, Inmos Ltd. [65].*

3.3.2 Internal and external inter-processor communications

The transputer uses the channel concept for inter-process communication. A channel is a memory location either containing `EMPTY`, indicating that no process uses the channel for communication, or containing a pointer to the workspace of the process that is using it for communication. Assume that two processes A and B wish to communicate with each other through a channel. Each local I/O request results in loading the required number of bytes to be transmitted (count), loading the pointer to the data that has to be transmitted, and checking the channel location, which initially is empty. The first process performing an I/O request (say process A), finds the channel location empty and stores its identity (pointer to process A) in it (Figure 3.10).

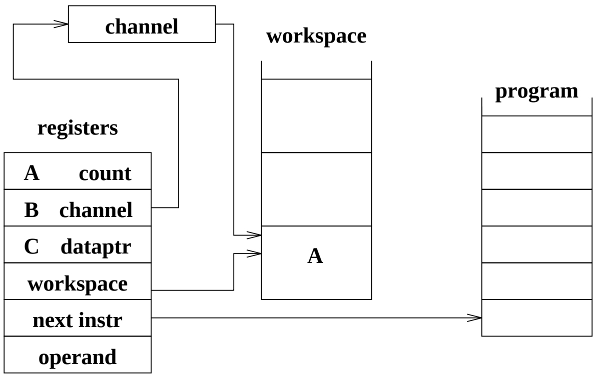


Figure 3.10: *Initiating an internal I/O request, process A is just before becoming inactive and waiting for process B's I/O request.*

Then, process A becomes inactive as it waits for I/O and subsequently becomes descheduled resulting in storing the data pointer, count and process context in its workspace. Whenever the second process (B) is scheduled and performs its I/O request, it finds the corresponding channel location occupied by the pointer to process A and is able to access A's data address through the previously stored data pointer (Figure 3.11). The actual communication is then performed via a block move and the inactive process A is added to the active process workspace list. Finally, the channel location is reset to its empty state. For this procedure, it does not matter which process performs the I/O request first.

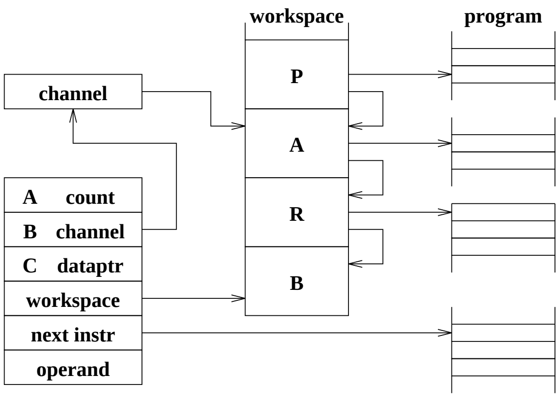


Figure 3.11: *Internal communication when process B has become scheduled.*

If a process wishes to communicate with another process running on a different processor over an external channel (i.e. transputer link), the scheduler recognizes the corresponding I/O request and delegates the task of communication to the corresponding link interface, while descheduling the process. The link interface administrates the descheduled process and recognizes starting address and count of the data to be transmitted. Each of the four link interfaces has three registers to store this information (see Figure 3.12). Furthermore, each link interface has DMA access to the transputer memory, so the CPU is not needed to copy the data. Similar as described above, processes initiate their corresponding channel for external communication, but in this case the channel is implemented through the link interface. When the link interfaces on both processors have initiated the transmission, the data is transmitted over the transputer link. If the data transfer is completed, each link interface places the corresponding process at the end of the active process list. Again, it does not matter which of the processes performs the I/O request first.

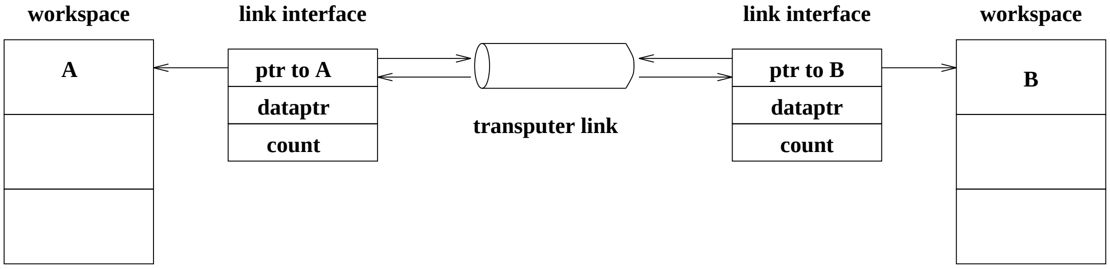
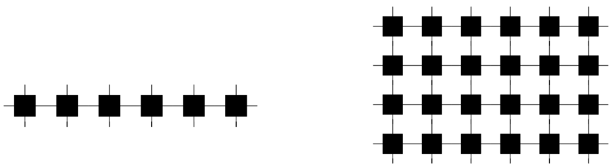


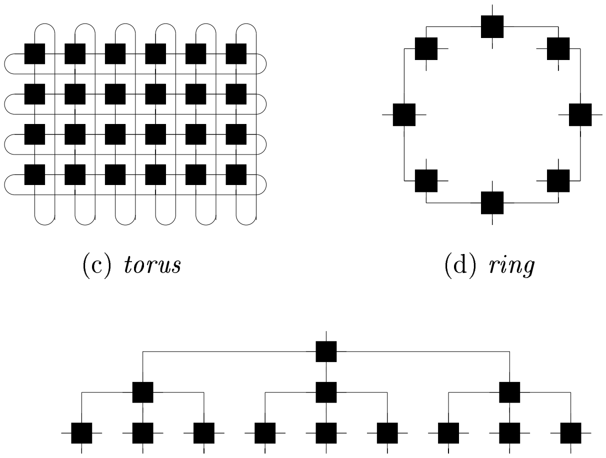
Figure 3.12: *External communication over a transputer link.*

3.4 Transputer networks

Transputer networks contain a number of transputers connected in a network via transputer links. As each processor only has four links, the connectivity of the network is restricted to four. The major advantages of using processors like the transputer as a building block for large parallel systems are that scaling up the number of transputers not only increases the computation power, but also increases the total communication bandwidth of the system, as for each transputer an extra four communication links are added. This means that conceptually no global communication contention or bottlenecks occur like is known from shared memory or shared buses. Depending on what topology suits an application best, several multi-transputer networks can be made like a pipeline, grid, torus, ring and tree:



(a) *pipeline* (b) *grid*



(c) *torus* (d) *ring*

(e) *tree*

Figure 3.13: *Some multi-transputer topologies. A transputer is depicted as a square box with four lines representing the links in the north, east, south and west direction.*

Connecting transputers via their links can be done in a fixed or reconfigurable topology. The latter is implemented by using programmable crossbar switches that allow to configure the transputer network in any desirable topology. Note that for fixed topologies the transputers are directly coupled via a transputer link, whereas when using crossbar switches, this is not the case. What quantitative implications this has for inter-processor communication delays will be discussed in a later section. The GCel-512 is configured in a 16x32 grid with fixed topology. The NSC (Nijmegen Super Cluster) uses the Inmos COO4 crossbar switches for implementing a reconfigurable system (Figure 3.14). A COO4 crossbar switch contains 32 demultiplexers that are able to couple any of the input links to one of the output links. COO4 chips can be cascaded to connect multi-transputer units with a higher degree of connectivity. The next discussion gives an overview over the architecture of the NSC, consisting of 64 transputers and several communication units consisting of COO4s and control logic.

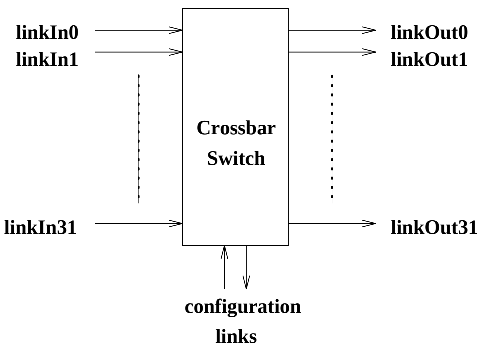


Figure 3.14: *The COO4 provides a switch between 32 input and 32 output links.*

3.4.1 Hierarchical architecture of the NSC

The Super Cluster [39] architecture has a hierarchical design consisting of basic building blocks, multi-transputer modules, network configuration units, computing clusters and multiple superclusters. The basic processing element in the NSC consists of a T800 transputer, 4 Mb of RAM and some control logic. Any processing element can connect to four neighboring processing elements. A multi-transputer module contains four of these processing elements (see Figure 3.15).

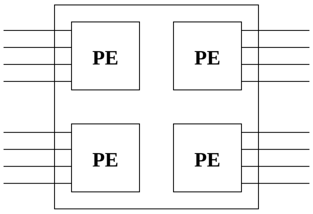


Figure 3.15: *The multi-transputer module MTM-EDC.*

A computing cluster contains four multi-transputer modules (MTMs) and a so called network configuration unit (NCU), which connects all 4x4 transputer links of each of the four MTMs (so the 64 links of the processing elements in a computing cluster connect to its NCU). Each NCU can connect 96 to 96 transputer links. This means that each computing cluster has 32 transputer links that can connect to other computing clusters or processing elements (see Figure 3.16).

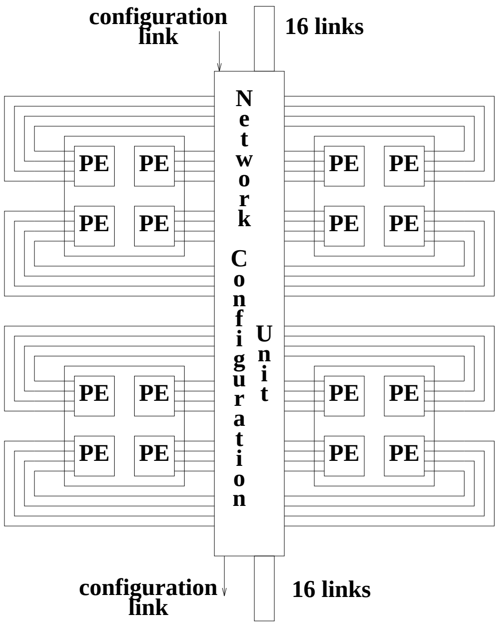


Figure 3.16: *The architecture of a computing cluster.*

On the top level hierarchy, the supercluster consists of one to four computing clusters. The computing clusters are connected and configured via 2 extra network configuration units. Via the latter, multiple superclusters may be configured in e.g. 256, 512 or 1024 multi-transputer systems.

3.4.2 Hierarchical architecture of the GCel

The GCel is one of Parsytec's GC family of transputer systems. GC means Grand Challenge, GigaComputer or GigaCluster (opposed to SuperCluster). The architecture of the GC family is designed with the goal of building highly scalable massively parallel transputer systems from building blocks of one GC containing 64 processors. Similar to the supercluster architecture, a GC contains 4 clusters which each contain 16 processing elements. Each processing element can be any Txxx transputer, but the main design goal was to build a highly scalable parallel machine based on the T9000 transputer with the C104 dynamic crossbar switch. Unfortunately, Inmos had serious problems with building these components and therefore current GC architectures are based on the T805 processor and C1004 crossbar switch. These systems are called GCel.

As with the supercluster, the GC has a highly modular architecture. What is called a computing cluster in the supercluster architecture, corresponds with a *cluster*. Each cluster has 16+1 processing elements, the 17th is used for redundancy purposes. Whenever one of the 16 processors crashes, the 17th takes over its control. Each four clusters together form a *cube*, which is the basic element of larger machines. The GCel-512 consists of 8 cubes, providing a total of 512 processors.

The cluster architecture resembles that of the computing cluster, however instead of one NCU, four C104 chips are used. The main reason for taking more crossbar switches is because the eventual goal of GC machines is to scale up to 16K processors which requires a very high global connectivity. Each cluster in a large GC network is connected to its six neighbors in a three dimensional grid via 8 links in each direction. Via the links, three communication networks are realized in a GC system. The *data* network provides the communication pathways for applications and the operating system. The *control* network connects the control processors of each cube. Via the control network, processors can be initialized or booted, routing chips can be programmed, and monitoring of the hardware status of the cube resources can be performed. The third network is the *I/O* network. This builds a way to connect transputers with I/O devices, like a host machine and memory storage devices.

3.4.3 Configuring a transputer network

For the GCel machines, a user or programmer need not to be concerned with how to connect the diverse transputer links, as they are "hardwired" in a grid. However, they can be configured in different ways by a system administrator. A user can only allocate partitions of a GCel which the administrator has installed. For the NSC, a configuration procedure is defined that allows a user to configure a transputer network in any topology that can be made with the four links each processor has. Using the operating system Helios (see Section 3.6), each user has to initiate a login procedure which performs password authentication and subsequently reads a *resource map* defining the topology and resources of the transputer network the user wishes to claim. If the required resources are available, Helios boots each transputer in the specified network and loads a copy of the distributed operating system, or – if the user has marked a transputer to run in `NATIVE` mode, leaves the processor empty. This allows for native programs or runtime systems to be loaded onto the processor without having Helios running on it. Each resource map consists of a number of *terminals*, where each terminal can specify a transputer or any hardware device like a frame grabber or graphics device. For each communication link of a processor it can be specified to which processor it has to be connected. As an example, consider the specification of a ternary tree of processors of depth 3:

```
subnet /Cluster {
  CONTROL Rst_Anl [/Cluster/00];
  terminal 00 { ~IO,,~01,; SYSTEM;
                ptype T800; Mnode Rst_Anl [pa_ra.d]; }
  terminal 01 { ~00,~02,~03,~04; HELIOS; ptype T800; }
  terminal 02 { ~01,~05,~06,~07; HELIOS; ptype T800; }
  terminal 03 { ~01,~08,~09,~10; HELIOS; ptype T800; }
  terminal 04 { ~01,~11,~12,~13; HELIOS; ptype T800; }
  terminal 05 { ~02, , , ; HELIOS; ptype T800; }
  terminal 06 { ~02, , , ; HELIOS; ptype T800; }
  terminal 07 { ~02, , , ; HELIOS; ptype T800; }
  terminal 08 { ~03, , , ; HELIOS; ptype T800; }
  terminal 09 { ~03, , , ; HELIOS; ptype T800; }
  terminal 10 { ~03, , , ; HELIOS; ptype T800; }
  terminal 11 { ~04, , , ; HELIOS; ptype T800; }
  terminal 12 { ~04, , , ; HELIOS; ptype T800; }
  terminal 13 { ~04, , , ; HELIOS; ptype T800; }
  terminal IO { ~00; IO; }
}
```

Algorithm 3.1: *Resource map specification for a ternary tree of depth 3. Each processor must run Helios and must be a T800 transputer.*

During the boot phase, the reconfigurable crossbar switches are programmed such that the required link configuration is installed.

3.5 New transputer systems

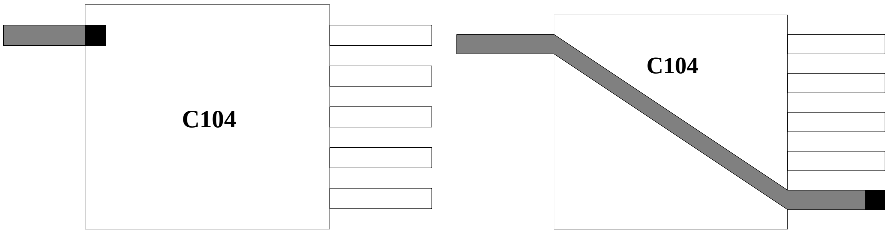
A wide variety of MIMD systems that communicate via a message passing communication network exists. Some of these contain processing nodes that provide a similar performance as the transputer, others have more up to date nodes with increasing computational powers. Current transputer systems based on the T4 or T8 series have proven their value in parallel research, applications and industry. However, compared to a current state of the art workstation or PC, the transputer has become very slow. In many cases, it just is not efficient enough to use a transputer system. In this section, the efforts of Inmos and Parsytec to boost the performance of a processing node or to build new communication chips for speeding up the communication network are briefly discussed.

3.5.1 T9000 based systems.

Since several years, Inmos has announced the arrival of a new transputer, the H1 or T9000. This processor would work in conjunction with a new communication chip, the C104. From the specifications of the T9000 [67], it provides a performance of 200 MIPS, 25 MFLOPS, runs at 50 MHz and each of its transputer links has a communication bandwidth of 20 Mb/second. This means that the computational and communication performance is increased by a factor 6 to 10 compared with current transputer systems. Apart from its increased performance, the main differences with previous transputer are:

- ◇ 16 Kb cache memory, which allows for applications to make better use of fast internal memory.
- ◇ Pipelined, super scalar architecture, which allows for several instructions to be computed simultaneously. The architecture also features an instruction grouper which recognizes instruction sequences that the processor can execute effectively.
- ◇ Better error handling and protection, which allows for the protection of each process' code and data locally.
- ◇ Memory management, which allows to check all memory accesses. Furthermore, it can be used to implement swapping from memory to and from disk.
- ◇ Virtual hardware channels, which multiplex a physical transputer link to be shared by multiple processes. Messages are divided in packets with a header containing information about the destination process.
- ◇ C104 routing chip, which allows for virtual point-to-point connections between processors that are not interconnected. The C104 uses an extra destination field in the header of a packet to determine whether it is destined for the local processor or it has to be trough routed to some other C104. A C104 contains a 32x32 crossbar chip, two control links and a local processor to execute routing tasks.

The C104 uses worm-hole routing to minimize routing delays. Instead of a store and forward mechanism, via worm-hole routing it is possible to output a packet while it is still being inputted. Upon receiving a packets header, it is decided to what link it has to be trough routed (see Figure 3.17(a)). After setting up the crossbar switch, the data is directly transmitted over the switch until the end of the packet is detected, after which the switch is cleared (see Figure 3.17(b)). Note that with this system, routing becomes transparent for the programmer and furthermore, the routing administration no longer has to be executed by the transputer as this is performed by the C104.



(a) *Determination of output switch.* (b) *Transmission.*

Figure 3.17: *Expected and measured times for* `GAB()` *on different GCel grids.*

In [114], it has been discussed what implications new transputer systems based on the T9000 and C104 communication chip could have on applications like parallel neural network simulations. The main idea is that – because the performance model splits computation and communication times – the new overall time to compute an application would be the sum of the computation time on a T8 transputer divided by the increased computational power and the communication time divided by the increased communication power. We were not able to quantitatively validate this expectation because no systems using T9000 transputers are available. Similarly, the C104 chip has not been released until now. There exist T9000β transputers which operate on a lower clock rate (10 MHz), which were tested [3], but because of the late arrival of the T9000, users and manufacturers are looking at alternatives, like the PowerPC.

3.5.2 The PowerXPlorer

The PowerXPlorer is a MIMD system shipped by Parsytec based on the PowerPC 601 [50]. The system uses transputers and their communication links to build a communication network. The PowerPC is a state of the art RISC processor developed by IBM, Motorola and Apple. It is a super scalar processor executing three instructions per clock tick (via integer,branch and floating point units). It contains a 32 Kb cache and an on chip memory management unit. Via its system interface (32-bit address bus, 64-bit data bus and 52 control and information signals), it can exchange data with other devices through shared memory.

The PowerXPlorer contains up to 64 PowerPC 601 nodes, each equipped with a T805 (1 Mb memory) processor dedicated to communication over its 4 links. Each node gives a peak performance of 80 MFLOPS double precision operating at 80 MHz and using the communication network, each node has 80 MBits/second communication throughput. Each pair of PowerPC and T805 communicate via an internal memory bus through shared memory, and can be considered as a single processing element. The system can be managed by the Parix operating system, thus allowing fast ports of applications from the GCel to the PowerXPlorer.

We were able to use the PowerXPlorer at the University of Amsterdam to examine the performance of parallel neural network simulation programs. This system contains 32 nodes, configured in a 2D grid topology. Users can claim networks of sizes 4x2, 4x4, 6x4 and 8x4 processors. Each node contains 32 Mb of memory. The system architecture comprises several units containing 2 boards with each 2 pairs of T805 and PowerPC. The 32 processors are contained in 8 units of 4 processing elements configured in a 2x2 grid. Parsytec is also incorporating this technology in the GC series, building large systems with tremendous compute power. However, especially in communicate intensive applications like neural networks, it will become clear that the communication performance should be increased too. Using a communication network based on C104 routing chips would facilitate such a resource.

3.6 Programming environments for the transputer

As any other computer, the transputer can be programmed on different levels of detail. Whereas most computers use an operating system to manage I/O devices, disks, screens and to provide a shell within which a user can issue commands to be executed, for many applications the transputer is used in so-called native mode. In this mode, it is ready to accept any executable code to be loaded and run, which is supported by a number of machine language constructs. The software that runs on a transputer and builds an intermediary between a user and the transputer system can be distinguished in three levels of detail, 1) low level application code, 2) runtime systems and 3) operating systems. Software development and compilation can be done on a host system, in which case the executable code is down loaded from the host onto the transputer system. Or software can be developed on the transputer itself, which involves that code compilation and linking is done on the transputer.

3.6.1 Native systems

A transputer runs in native mode when the first two levels are concerned. This involves that a programmer has to take account for problems like (global) inter-processor communications, task scheduling and task distribution. This may be supported via several software layers providing specific algorithm libraries. This software can provide a programmer with various software development features, as listed in Table 3.1

A wide range of programming languages has been developed for the transputer, like ANSI C, various versions of C with parallel language constructs, Occam, C++, Ada, Fortran, etc.

In general, the parallel language constructs they offer are `PAR` and `select` statements and some `CHANNEL` data type, which represents a communication medium between tasks. The `PAR` construct allows for a programmer to start up concurrent processes on one processor (like `fork/exec` in Unix).

parallel languages	parallel language constructs
	compiler, linker, libraries,
	tools for creating and loading parallel programs
tools	profiling/monitoring tools,
	multi-tasking/multi-processor debuggers
libraries	vector/matrix operations,
	routing and communication routines,
	signal processing, image processing,
	graphics libraries,
	string operation libraries,
embedded systems	routines for DMA access, interrupt routines

Table 3.1: *Native software development features.*

Using the `CHANNEL` datastructure, point-to-point communication between tasks can be established. If two communicating tasks run on distinct processors that are not directly connected, the message to be communicated has to be routed through the transputer network, as is explained below in Section 3.6.4. The `select` statement is used to determine which of a number of communication channels becomes active. In general, this is very useful for asynchronous applications like a farm construct. In a farm, a master process distributes the available work over a number of slaves. The work is divided in packets and initially, each slave gets one packet. The moment a slave becomes ready, it sends a message to the master requesting for more work. In order for the master to determine which slave becomes ready, the `select` statement can be used. A simple way to program such a farm is depicted in algorithm 3.2

```
CHANNEL master, slave[NSLAVES];
void master (int id)
{
    SendInitialPackets();
    while (GetNewPacket(&packet))
        select {
            alt (i=0;i<NSLAVES;i++)
                guard slave[i]: SendNewPacket(i);
                break;
        }
}
```

(a) *Master.*

```
void slave (int id)
{
    while (1) {
        ReceiveNewPacket();
        HandlePacket();
        send(master,READY);
    }
}
```

(b) *Slave.*

Algorithm 3.2: *Farm implementation (ParC syntax)*

3.6.2 Task distribution and execution

Note that in algorithm 3.2, both the slave process and master process could run on the same processor, or several slaves could run on one and others on distinct processors. However, in general it would be best if all tasks would run on a separate processor, as in such a case all tasks can be really executed in parallel. Now for the farm problem at first sight it does not matter on which processor a slave is running. As all packets have the same size and all slaves perform the same computation, the time for each slave to compute one package is equal —-, if all processors provide the same computational power. For problems that do not have such a regular structure, it can become important to specifically map each task on a specific processor. There are three ways in which a task can be loaded onto a processor:

1. Each task is programmed and compiled into a separate piece of code. The code is explicitly loaded onto a specific processor. For example the Helios operating system supports this way of task distribution.
2. Each program contains the code for all tasks and is loaded onto every processor. It is decided at runtime which piece of the code has to be executed. For example for many native systems tasks are loaded on a processor via this mechanism.
3. A small piece of code is loaded onto each processor. On a processor it is decided at runtime which task has to be executed, after which the separately compiled code of the task is loaded. For example the Parix operating system supports this way of task distribution.

Note that using operating systems like Helios and Parix, the second method of loading tasks on a processor can also be exploited. In the second and third method, when running code on a processor it is decided what task has to be executed. This is realized by examining a datastructure that specifies the transputer configuration and the transputer on which the code is running. For example, using the Parix `RootProc_t` structure, on each processor it can be determined what grid size the GCel has and what position the processor has in the grid.

```
typedef struct {
    int MyProcID;                     /* own processor number */
    int MyX; int MyY; int MyZ;        /* (x,y,z) position */
    int nProcs;                       /* DimX * DimY * DimZ */
    int DimX; int DimY; int DimZ;     /* (x,y,z) dimension */
} RootProc_t;
```

A very general application of using such a structure is in master-slave programs, where the master process must reside on a specific processor and the slave processes on the other processors:

```
if (rootproc->MyProcID==0)
     master();
else
     slave();
```

Using an operating system provides a relatively comfortable way of programming, task distribution, scheduling, routing and debugging. Furthermore, similar features are supported by an operating system as the ones listed in Table 3.1. Many operating systems developed or ported for the transputer exist, such as CSTools, Express, Idris, Mach, and Chorus. For the experiments discussed in this thesis, the operating systems Helios and Parix were used.

The way in which tasks are distributed when using Parix is discussed above. Helios provides another mechanism based on CDL-scripts. CDL (Component Distribution Language) [108], is a Unix shell-like language that is used to specify a so-called taskforce. This is a description of a parallel program consisting of multiple tasks (components) which communicate to each other by message passing. The communication network can be specified via predefined communication constructs or by explicitly specifying each pair of inter-task connections. When running a CDL taskforce, if it is specified correctly, the components are mapped on the required processors. By annotating each component with the identification of the processor to run it on, components can require to be run on a specific processor. Similarly, other requirements like the required available memory per processor can be specified in a CDL-script.

The next CDL-script specifies three components `A`, `B` and `C`, which are connected in a pipeline. When running this script, for example component `A` runs the code `a` on processor 0, sends its output to component `B` on processor 1, which sends its output to component `C`.

```
component A { code a; puid 00; }
component B { code b; puid 01; }
component C { code c; puid 02; }

A | B | C
```

Algorithm 3.3: *CDL-script containing three components.*

Conceptually, the Helios philosophy works well. When developing a parallel program consisting of multiple tasks, it can be really helpful to be able to specify a taskforce and in particular its communication structure using CDL-scripts. Also the mapping of tasks on processors can be handled on this level. However, the determination of the communication channels for each component must confirm to the CDL-script. How this can be realized, is discussed in the sections below.

3.6.3 Setting up communication channels

Both Helios and Parix provide several levels of setting up communication channels between tasks. In Helios the channels can be set up via a CDL specification (which implements channels as Posix file descriptors). Each task program should know the meaning of a file descriptor, i.e. to which task it is connected and what amount of data it must communicate at a certain moment in execution. In Figure 3.18, an example assignment of file descriptors to channels following a CDL script is depicted.

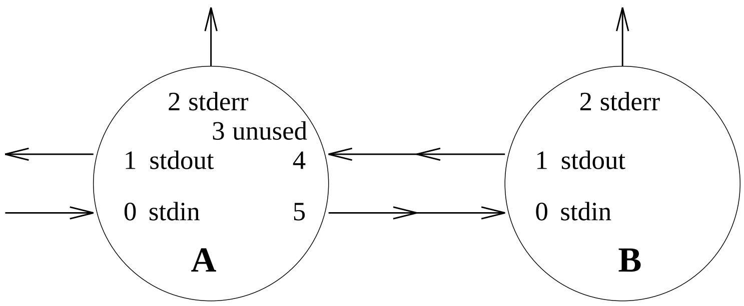


Figure 3.18: *Assignment of file descriptors for A <> B.*

By using posix `read` and `write` calls, two connected tasks can communicate to each other. Note that the posix channels are so called virtual channels, as two communicating tasks could be running on any processor, and their communication paths could thus consist of many physical inter-processor communications. This means that using CDL-scripts, it is possible to write parallel software independent of the underlying target hardware, where routing of messages is performed by Helios.

Parix provides a set of communication libraries for setting up virtual communication topologies like a ring, grid, tree or torus. For example, using the routine `Make2DGrid` returns a structure containing the four communication channels in the `NORTH`, `EAST`, `SOUTH` and `WEST` directions. After investigating the position in the topology, each process can access the corresponding channel by indicating the direction it is interested in. Similarly, processes can communicate in for example a ring by indicating the `FORWARD` and `BACKWARD` directions.

Like with Helios, this way of using virtual channels allows for writing software independent of the hardware topology. However, especially for applications where a lot of communication is involved, there can arise a severe cost penalty if the hardware topology does not match that of the virtual communication network. For example, mapping a ternary tree of depth 3 (40 processors) on a 5x8 grid results in an inefficient organization. Therefore, for the experiments used in this thesis, a communication library was designed and implemented that explicitly maps a tree onto a tree topology and a grid onto a grid (see chapter 5). This software layer uses point-to-point communications between processes, where each process communicates over physical communication links to its neighboring processes.

3.6.4 Inter-processor communication and routing

After booting, for any distributed operating system like Helios or Parix, a daemon process is running on each processor. This process, which in Helios is called the *nucleus*, loads and schedules tasks on a processor and enables (inter-processor) communications between tasks. Communication over processor links is handled via a router mechanism which monitors for each processor the incoming links and investigates whether an incoming message is designated for the processor or whether it has to be through-routed to other processors. In general, routing is performed in two steps. In an initial step, the processor network is investigated and for each processor a so-called routing table is created. Using this table, during the second step – which is when running the system – the router process can determine via which link a message has to be through-routed to a destination processor.

Whereas an operating system provides (through virtual channels) an implicit routing mechanism, for native systems routers have to be explicitly implemented. For the ParC runtime system, a general router was designed in our department [90], which supports broadcast and gather operations like the ones used in parallel neural network simulations. Similar routers are designed for other systems, like Tiny for Occam [19]. In general it can be stated that software routers require some overhead because they must implement a general mechanism including the routing tables and buffering of messages. If the patterns of communication for a particular application or range of applications are well known, it is better to use a communication layer which is tuned for the application and that implements global communications as efficient as possible. In Chapter 5, the design and implementation of such a communication layer is discussed.

4

A Scalable Performance Prediction Model

Outline

A method to determine the suitability of transputer systems for parallel neural network simulations is introduced in this chapter. By predicting the performance of such systems, suitability issues like how fast the neural network can be run and how efficient the execution platform used is, can be answered. Two existing methods for predicting the performance of an execution platform are discussed: performance benchmarking and performance modeling. It will be made clear that the first method can only be used to give rough estimates and that questions regarding speedup and scalability cannot be answered. Examples of the second method will indicate that if an application is modeled on a too fine-grained level (i.e., on the level of arithmetic operations), the resulting performance timings are not precise. A combination of performance modeling and performance benchmarking on the level of kernel benchmarks will be presented in this chapter. The new combined method achieves much better prediction results.

The performance of an execution platform for neural networks is often expressed in MCUPS or GCUPS (Million or Giga Connection Updates Per Second) and MICPS or GICPS (Million or Giga InterConnects Per Second). The achieved performances reported in the literature differ enormously, ranging from a couple of thousand ICPS for workstations to more than two GCUPS for VLSI hardware [43, 78]. But what does such a performance scale mean to the practical user. Does it provide any help when deciding which execution platform would be the best for a specific application? Do MCUPS say anything about the quality of the implementations? Are the efforts for programming and debugging the machines mentioned? Or can the performance of another platform for the neural network or the performance of the platform for another neural network algorithm be predicted based on this information? In fact, all these questions cannot be answered by only using a performance scale like MCUPS. What is needed is a means via which a computer system can be evaluated for a specific application.

The evaluation of computer systems for a certain problem can be performed from four different points of views [54]: (1) how fast can a given problem be solved using a system (responsiveness), (2) how efficiently is the system exploited when solving the problem (usage level), (3) how well can the system deal with failures and other unexpected events (mission ability and dependability), and (4) what support does the system offer to a user or programmer for realizing the problem (productivity). The first and second point of views will form the main contents of this part of the thesis. It will appear that by answering the first question, we can also answer the second. By computing the performance of a system for an application, also the efficiency and speedup can be computed. Three methods for performance evaluation will be discussed in this chapter: performance benchmarking, performance modeling, and a hybrid combination of benchmarking and modeling.

4.1 Performance Benchmarking

There are two major reasons why benchmarks are important. Manufacturers can run the same benchmark on different machines to compare their CPU, memory or communication performances. And benchmarks can be run on one specific machine to quantify its individual performance for e.g. floating-point operations, memory accesses, numerical libraries or application areas. If one would want to know how well an execution platform would be suited for a particular application, one could select the benchmark which best matches the application and consider its results for that specific platform. Benchmarking methods can be divided into a number of categories [5, 118], which are synthetic, kernel, algorithm and application benchmarks.

4.1.1 Synthetic benchmarks

Synthetic benchmarks are not representative of any real computation but exercise various basic operations, such as memory latency, library routines, arithmetic operations, etc. Benchmarks that fall into this category are the well known Dhrystone [117] and Whetstone

[21] benchmarks. The Dhrystone benchmark contains a collection of statements from non-numeric, system-type programs, such as use-interfaces, operating systems, compilers and editors. The performance of Dhrystone depends on cache size, but the code fits in the cache of most modern machines. Furthermore, the implementation of the string functions influences its performance significantly. The Dhrystone benchmark comprises language statements occurring in numeric application programs. These contain more loops, more floating-point operations, more numeric library functions, less procedure calls and less conditional statements. The benchmark contains a number of small loops with a high code density resulting in near 100% cache hits, which obviously does not hold for real applications.

This highlights the major disadvantage of synthetic benchmarks, i.e. they do not mirror real applications because of two reasons. First, they do not really test the memory system because in general the code and data sizes are very small compared to real sized problems. This involves that small sized synthetic benchmarks can be loaded in fast internal memory or on-chip cache, whereas real applications cannot. Second, compiler writers tend to boost their compiler performance dedicated to the typical operations that are contained in e.g. Whetstone or Dhrystone benchmarks, such as numerical library routines and string operations. This could imply that the compiled code for real applications is not that efficient.

4.1.2 Kernel benchmarks

Kernel benchmarks eliminate these problems as they are more representative for real applications. These benchmarks consist of program parts that embody the salient features of compute, communication or memory extensive portions of actual applications. Kernel benchmarks can be easily ported and measured on other target platforms, as they contain compact programs compared to large benchmarks or full applications. Examples of kernel benchmarks are discussed by for example Berry *et al* in [5]. They mention The Livermore Fortran Kernels and NAS kernel benchmark program. The main idea behind kernel benchmarking is to use the performance figures of the kernel that has similar memory usage, uses the same library routines and typical numerical operations as the application for which the performance has to be indicated. Typically, the benchmarks can be scaled to measure the required memory access behavior as imposed by the application. Kernel benchmarks contain complete routines that are stripped from real applications or numerical libraries, unlike synthetic benchmarks which merely contain statements or pieces of such packages.

4.1.3 Algorithm benchmarks

Algorithm benchmarks contain sub-programs that implement familiar algorithms well known from the literature, as found in image processing, numerical, statistical or other application areas. Examples are algorithms found in Maple [31], Matlab [66], FFT libraries and numerical algebra libraries. Similar to kernel benchmarks, comparing an application with algorithm benchmarks matching its costs will indicate its performance range that can be achieved. An example of algorithm benchmarks is Linpack [26]. In general, algorithm benchmarks can be scaled by increasing or decreasing the problem size, e.g. the size of the matrix on which Linpack operates. Algorithm benchmarks for scientific applications are characterized by the high amount of floating-point operations, loops, and high locality of code compared to the locality of data. The latter involves that target architectures with fast memory access and instruction cache are in favorite.

4.1.4 Application benchmarks.

Although kernel and algorithm benchmarks try to match a wide range of application areas, they are still hindered by the effects of cache size, memory access times, compiler optimizations and implementations of library routines. It has been argued by a number of authors [5, 24, 105, 118] that for reflecting the performance for real applications, small benchmarks are not sufficient. Rather than quantifying the performance of individual system components, a wide range of *application*-oriented benchmarks are required, which give a more resembling cost profile than kernel or algorithm benchmarks. Application benchmarks are for example the SPEC [24], Perfect, Genesis [46] or EuroBen [105] benchmarks. Characteristic for these benchmarks is that each is initiated by a collaboration between research and commercial organizations with the goal to come up with a standardized set of application programs to be used as benchmarks for a wide range of target platforms.

4.1.5 Suitability of performance benchmarking

Our goal is to arrive at a benchmarking method for predicting the performance range of MIMD parallel processor systems for parallel neural network simulations. Two approaches can be considered for arriving at such a method. The first is to examine whether existing benchmarks like the ones described in the preceding sections can be used to quantify the expected performance range of new applications. The other approach is to find out whether it is feasible to develop synthetic, kernel, algorithm or application benchmarks that are scalable as well as general purpose. The second approach will be discussed in section 4.3.

Considering the first approach, it must be concluded that existing benchmarks can perfectly be used to measure the performance range of a target machine for the benchmark involved. Measuring the same benchmark on different platforms can be used to compare their performances. As it is likely that manufacturers of new machines will port most well-known benchmarks in order to present performance characteristics, in most cases the numbers that are listed in their specifications can be used for the comparison. However, they will publish only these numbers which are in favor for the machine, and probably will use all their skills to boost the performance of the benchmarks. There exist many factors that determine the performance of a benchmark, and they all must be taken into account to be able to use benchmark results as a performance indication for other applications like parallel neural network simulations:

1. The hardware platform, characterized by the number of processors, processor architecture, memory and communication system, and peripherals.

2. The software environment, comprising operating system, programming languages, compilers and libraries.

3. The size and complexity of the application.

4. The implementation of the application and optimization tricks used.

5. Communication and synchronization requirements.

Consider that the number of operations required for updating a connection depends on the complexity of the neural network model. For example, comparing MCUPS measured for a Hopfield network [49] with those measured for a backpropagation network [83], will give significant performance differences, as updating a connection for the latter network not only requires more operations, but also uses expensive library functions such as `exp()`. This is one of the reasons why we state that MCUPS is a meaningless performance scale. It would be appropriate to express the performance in Hopfield MCUPS or backprop MCUPS. Furthermore, if variations in an algorithm exist, these should also be indicated when stating performance, like *[Rumelhart,momentum,batch update]* backprop MCUPS.

To my knowledge there are no performance benchmarks containing neural network codes. Furthermore, if there would exist such benchmarks, they cannot cover all variances in neural network models that exist. This involves that when using such benchmarks only rough estimates of the performance that can be expected can be given. So if one wants to predict the performance of a particular application, and use the performance number of the benchmark that best matches the application as an indication, a number of criteria have to be taken into account in order to make this number a reliable estimate. These criteria involve that if this approach is taken, the selected benchmarks have to be examined in great detail:

1. Make sure that the benchmark has been measured with the same compiler and optimization settings that will be used for the application.

2. Make sure that the same programming language and library routines are used.

3. Make sure that the benchmark is programmed in the same programming style as the application will be implemented.

4. Take the performance numbers for benchmarks that have the same problem size as the application.

5. Consider the number of optimization efforts that have been taken to port the benchmark.

6. Make sure that the benchmark requires the same amount of synchronization and communication overheads.

4.2 Performance Modeling and neural networks

Another approach to predict the performance of an parallel neural network implementation is to analyze its complexity in terms of the required communications, data accesses and arithmetic operations. Given the costs for these basic operations, quantitative estimates about the performance that can be achieved could be given. Most performance analyses reported in the literature either analyze a particular neural network algorithm and target machine architecture [18, 124, 125] or use some general model characterizing the architecture and functioning of a whole range of algorithms and platforms [38, 69, 61]. For the discussion of performance analysis models below, the performance parameters characterizing a neural network and target platform are listed:

n	total #neurons	t_a^l	time for local memory access
w	total #connections	t_a^g	time for global memory access
P	total #processors	L	#communication links per processor
P_i	processor #i	B^L	bandwidth per communication link
C_i	neural network part on P_i	t_t	transfer time for communication
S_{ij}	j-th sub-part of C_i	t_s	setup time for communication
λ^a	#local/#global memory accesses	λ^c	#local/#global communications

Table 4.1: *List of parameters often used in performance analysis models.*

Most of these parameters have to be known in order to determine the total costs for a parallel implementation. The overall cost for a MIMD parallel implementation for a network specified by NET decomposed over P processors is defined as:

$$t(P, NET) \quad = \quad t_{calc}(P, NET) + t_{comm}(P, NET)$$

In the following discussions, a further analysis of t_{comm} and t_{calc} is given for MIMD parallel processor systems.

4.2.1 Communication costs for neural networks.

When decomposing a neural network over a processor network, the goal is to ensure that each processor has an equal amount of work to do, and that the amount of inter-processor communications is minimized. Let the network be decomposed in P components $C_1 \cdots C_P$, and let each component C_i consist of a number of N_i sub-components $S_{i1} \cdots S_{iN_i}$. It will be evident that finding partitions with equal work loads is a major problem if the neural network is structured in modules with highly different computational requirements. However, assuming that the load is well balanced, the calculation time on P processors can be defined as $t_{calc}(P) = t_{calc}(1)/P$.

The following discussion presents a model in which the communication requirements for a given neural network on a MIMD machine architecture are quantified, as adapted from a model presented by Ghosh and Wang [38]. Given a partitioning:

$$\begin{aligned}(P, NET) \;=\; \{ \; & \{C_1 = \{S_{11} \cdots S_{1N_1}\}\}, \\ & \{C_2 = \{S_{21} \cdots S_{2N_2}\}\}, \\ & \quad \vdots \\ & \{C_P = \{S_{P1} \cdots S_{PN_P}\}\} \\ \} \; & \end{aligned}$$

Furthermore, assume that the basic amount of information that is transmitted is proportional to the size of a sub-component S_{kl}. Let c_{ij} denote the number of sub-components in C_i that have to exchange values with sub-components in C_j and let d_{ij} denote the number of inter-processor communications (hops) required to exchange information between processors i and j. The total number of hops can then be expressed as in Equation (4.1), where c_{ij} can be estimated as $\frac{\lambda^c \cdot n_i}{P}$, if a random distribution of connections (C_i, C_j) is assumed, so $\forall_{j,k} : \; c_{ij} = c_{ik} = \frac{\lambda^c \; n_i}{P}$:

$$\begin{aligned}\#hops &= \sum_{i=1}^{P}\sum_{j=1}^{P}(c_{ij} \cdot d_{ij}) \\ &= \sum_{i=1}^{P}\sum_{j=1}^{P}\left(\frac{\lambda^c \cdot n_i}{P} \cdot d_{ij}\right) \\ &= \frac{\lambda^c}{P} \cdot \sum_{i=1}^{P} n_i \sum_{j=1}^{P} d_{ij} \qquad (4.1)\end{aligned}$$

For MIMD processor systems, each inter-processor communication has to be initiated by fetching the data to be transmitted from memory, scheduling communicating processes, determining the output communication medium, etc. The costs required for initiating a communication are defined as the setup time t_s. Depending on the communication medium, the costs for transferring data are defined as the transfer time t_t. The total communication costs can be estimated from (4.1) where it is assumed that the basic amount of information that is communicated is determined by the largest sub-component as B bytes:

$$t_{comm} = \left(\frac{\lambda^c}{P} \cdot \sum_{i=1}^{P} n_i \; \sum_{j=1}^{P} d_{ij}\right) \cdot (t_s + B \; t_t) \qquad (4.2)$$

For homogeneous neural networks, it can be assumed that all neurons are equally divided over the available processors, so all n_i equal n/P. So Equation (4.1) can be rewritten as:

$$\#hops = \frac{\lambda_c \cdot n}{P} \cdot \sum_{i=1}^{P}\sum_{j=1}^{P} d_{ij} \qquad (4.3)$$

Note that in this equation each value is communicated individually, which in general is not done. Instead, all values from components C_i are sent in one packet, and thus the communication costs can be expressed as:

$$\begin{aligned} t_{comm} &= \frac{\lambda_c}{P} \cdot \sum_{i=1}^{P} \sum_{j=1}^{P} (d_{ij} \cdot (t_s + n_i \cdot B \cdot t_t)) \\ &= \frac{\lambda_c \cdot (t_s + \frac{n}{P} \cdot B \cdot t_t)}{P} \cdot \sum_{i=1}^{P} \sum_{j=1}^{P} d_{ij} \end{aligned} \tag{4.4}$$

If we let all d_{ij} equal the maximal path that has to be traveled in a processor network, i.e. the diameter d of the processor network, (4.4) can be estimated by the upper bound:

$$t_{comm} = \lambda_c \cdot d \cdot P \cdot (t_s + \frac{n}{P} \cdot B \cdot t_t) \tag{4.5}$$

Note that for a given number of processors, the communication time depends linearly on the number of neurons, whereas for a given number of neurons, the time is proportional to the square root of the number of processors. This is because the maximal path length on a grid is estimated as $d = \sqrt{P}$. For a ternary tree, the maximal path length is 2 times the depth of the tree, which is proportional to $d =^3 \log P$.

As will be explained in this thesis, for homogeneous neural networks in general all-to-all broadcasts are required. These networks have a regular architecture of one or more fully connected layers of neurons, so in general all neurons residing on a processor have to exchange their values with all neurons residing on the other processors. This involves that all λ_c are 1. In [69], it is explained that when not assuming fully connected layered networks, but instead using random connections between different network components, still all-to-all broadcasts are required. Assume two components C_i and C_j having equal sizes z, and having a connectivity λ^c, i.e. $\lambda^c \cdot z$ neurons in C_i and C_j need to communicate information. The chance that a given node in C_i has no connections to C_j is $(1-\lambda^c)^z$. This involves that even for relatively small z (say 100) and small connectivity λ^c (say .05), the chance that there is no communication required between processors i and j approximates zero. This means that for randomly connected neural networks there is always a need for all-to-all communications. The time required for these communications can be bounded by multiplying Equation (4.5) with the number of processors P.

The general model presented here, and the ones described in the papers of Ghosh and Wang [38] and Murre [69] discuss a general design specifying the communication requirements for parallel neural networks. When applying such a general model to a concrete neural network simulation, the corresponding parameters have to be estimated, analyzed or measured. Also, the general model must often be translated into models that match the application more specifically. For example Equation (4.5) specifies the maximal time required for a one-to-all communication. The parameters B, t_s and t_t have to be determined and furthermore, it has to be considered what the amount of inter-processor communications for a given parallel neural network simulation actually is in order to give a precise estimate. In the subsequent chapters several more specific models will be given that specify the required communications more precisely.

4.2.2 Computation costs for neural networks.

A number of efforts have been made to implement neural network simulations on parallel hardware and model their performance. These are all based on a careful examination and complexity analysis of the resulting simulation programs. In this section, several examples are given of methods that use such an analysis to model the performance of a simulation program in terms of the required arithmetic operations, memory accesses and communications primitives. For the discussion given below, the following parameters are used, where it is assumed that for each arithmetic operation the time needed for memory load and store operations is included:

t_{add}	addition	$+$	t_{sub}	subtraction	$-$
t_{mul}	multiplication	$*$	t_{div}	division	$/$
t_{sig}	sigmoid	$\frac{1}{1+\exp(-x)}$			

Table 4.2: *List of parameters used for arithmetic operations.*

4.2.2.1 The Kohonen SOM.

Remember from the introduction that the Kohonen SOM finds the best match between the neurons in the feature map and each input pattern $\vec{x}$. The "winning" neuron is the one with the best match and training the SOM for this pattern is done by updating the weights of the winner and its neighboring neurons. The operation of the SOM activation and training phase is depicted in algorithm 4.1.

```
load_data();
initialize_network();
while(err_crit>error && nepochs--) {
  for (all patterns p) {
    c = find_winner(p); /* activation phase */
    for (all neighbours i of c) /* training of weights */
      update_weight(i,p);
  }
}
```

Algorithm 4.1: *Algorithm for the Kohonen self-organizing feature map.*

Complexity of the SOM activation phase.

Given a pattern $\vec{x}$ with dimension N. For all n neurons, the Euclidean distance has to be computed following Equation (4.6):

$$d_i = \| \vec{x} - \vec{w}_i \| = \left(\sum_{j=1}^{N} (x_j - w_{ij})^2 \right)^{\frac{1}{2}} \tag{4.6}$$

Algorithm 4.2 depicts how the winning neuron is computed:

```
int find_winner (int pat)
{
        float *x,*w,dist;
        float min = MAXFLOAT;
        int i,j,winner;
        for (i=0;i<n;i++) {
                x = data[pat];
                w = weights[i];
                dist = 0;
                for (j=0;j<N;j++)
                        dist += (x[j]-w[j])*(x[j]-w[j]);
                if (dist<min) {
                        dist = min;
                        winner = i;
                }
        }
        return winner;
}
```

Algorithm 4.2: *The routine* `find_winner`*. In [58], finding the winning neuron is described through a process called relaxation. In this process, all neurons are connected via inhibiting lateral connections. Each neuron excites itself. Via relaxation, after a number of iterations, one neuron will become active (the winner), while the others remain inactive. Such a network is also known as winner-takes-all network.*

Note that for finding the winner, no computation of the square root is necessary. Computing the distance in `find_winner` requires $n \cdot (N \cdot (2 \cdot t_{sub} + t_{mul}))$. So the times for computing the activation phase for all patterns can be estimated as:

$$t_{act} = p \cdot n \cdot N \cdot (2 \cdot t_{sub} + t_{mul}) \tag{4.7}$$

Complexity of the SOM training phase.

For a pattern $\vec{x}$ and winner c, all neurons i laying within the neighborhood of c are updated following:

$$\vec{w}_i \quad += \quad \alpha \cdot (\vec{x} - \vec{w}_i) \cdot \eta(\sigma, c, i) \tag{4.8}$$

The weights are updated using the learning rate α and the neighborhood function $\eta(\sigma, c, i)$[1], which is determined by a Gaussian function whose strength is based on the distance between the winner c and a neuron i and the width σ (see Figure 4.1).

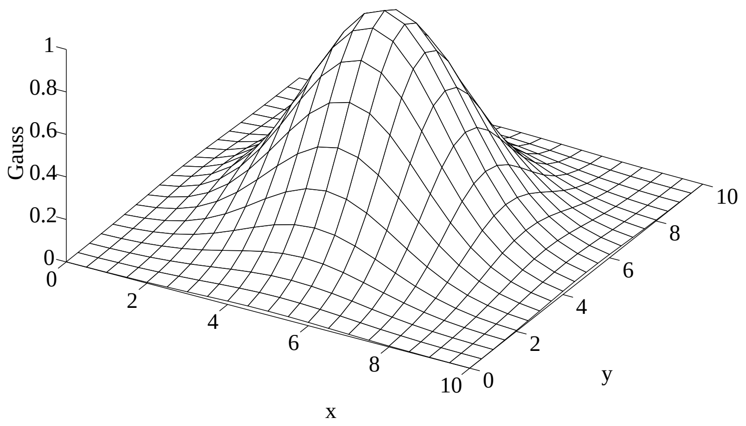


Figure 4.1: *Neighborhood function $\eta(\sigma, c, i) = \exp(-\frac{\delta(c,i)^2}{\sigma^2})$, in this case $\sigma = 2.5$, the winner $c = (5, 5)$ and the neurons i are located within the 10x10 Kohonen map.*

During training, the learning rate α and neighborhood parameter σ are decreased gradually. Effectively this means that the width and height of the Gauss function reduce during training. Though there exist many ways to perform this, in general for each iteration the value of α and σ is multiplied by a factor $k < 1$. Furthermore, the neighborhood range which determines the number of neurons that are updated, is decreased from an initial value (often $\sqrt{n}$) to zero. In general this is done by subsequently maintaining the range for a number of iterations, and then decrementing it. In practice this means that the neighborhood administration requires only to be computed in cases that the neighborhood range is changed, involving that the check whether neurons lay within each others neighborhood boils down to a lookup rather than a computation. These kind of programming tricks are known as the time/space trade off. In order to save execution

[1]There also exist variations of the algorithm that only use the learning rate.

time, it is often better to compute some results, store them in memory and use them during computation instead of computing them each time they are needed. Similarly, the results for the Gauss values have to be recomputed only when σ changes.

Analyzing the complexity of an algorithm where the computational load is not constant requires an estimation of the load. In this case, the load varies with the neighborhood range, as only the neurons that lay within this range have to update their weights. Furthermore, the load depends on where the winning neuron is located in the Kohonen map. If it is a border neuron, this number is obviously smaller than if the winner is at the center of the Kohonen map. Of course, it is possible to give a worst case for the complexity, but for performance prediction this could result in estimations with large deviations.

Wu, Hodges and Wang present a complexity analysis of the Kohonen SOM in [125]. They approximate the number of nodes in the neighborhood of the winner in the k-th iteration as:

$$n^k = \frac{\pi}{4} \cdot (2 \cdot r^k + 1)^2$$

The time to update the weights of all neurons laying in the neighborhood following Equation (4.8) would then sum up to $n^k \cdot (2 \cdot t_{mul} + 2 \cdot N \cdot (t_{add} + t_{sub}))$. In their implementation, only the learning rate α is used, so the computation of the Gauss values is not performed. Furthermore, the problem of computing whether two neurons lay within each others neighborhood is solved by just computing their distance, which for each pattern and each neuron i requires the computation of $(c_x - i_x)^2 + (c_y - i_y)^2$ and checking whether this is within the square of the neighborhood range[2]. This requires for each pattern $n \cdot (2 \cdot (t_{sub} + t_{mul}) + t_{add})$ computations. The total time for the k-th iteration of the training phase would then sum up to:

$$t^k_{train} = p \cdot (n^k \cdot (2 \cdot t_{mul} + 2 \cdot N \cdot (t_{add} + t_{sub})) + n \cdot (2 \cdot (t_{sub} + t_{mul}) + t_{add})) \quad (4.9)$$

and the total time for the training phase can be determined by integrating this expression for k.

4.2.2.2 The backpropagation neural network.

Algorithm 4.3 depicts the general operation of the backpropagation network.

[2]In [125], the square root of these values are is computed, which is not needed.

```
load_data();
initialize_network();
while(err_crit>error && nepochs--) {
  for (all patterns p) {
    /* activation phase */
    clamp_input();
    for (l=1;l<L;l++)
      compute_activations(l);
    /* training phase */
    compute_output_deltas();
    for (l=L-1;l>0;l--) {
      propagate_deltas_back();
      change_weights();
    }
  }
}
```

Algorithm 4.3: *The backpropagation algorithm (L=number of layers).*

Complexity of the backpropagation activation phase.

For each pattern $\vec{x}$ of dimension N, each of the N input neurons i is activated with the value x_i. The activations of the n_l neurons in subsequent layers l are formed by the sigmoid of their net input ξ_i:

$$\begin{aligned}\xi_i &= \sum_{j=0}^{n_{l-1}-1} (a_j \cdot w_{ij}) - \theta_i \\ a_i &= \frac{1}{1+\exp(-\xi_i)}\end{aligned}$$

The total time required for computing the state of activation of the network consisting of L layers is:

$$\begin{aligned}t_{act} &= \sum_{l=1}^{L-1} (n_l \cdot (n_{l-1} \cdot (t_{add} + t_{mul}) + t_{sig})) \\ &= w \cdot (t_{add} + t_{mul}) + (n - n_0) \cdot (t_{sig} + t_{sub}) \qquad (4.10)\end{aligned}$$

Complexity of the backpropagation training phase.

During training, each output neuron computes its errors as the difference between the target and computed output value:

$$e_i = t_i - a_i$$

Furthermore, it computes its so-called delta as the product of the error and the derivative with respect to the activation. The derivative of the sigmoid activation function $f'(a_i)$ equals $a_i \cdot (1 - a_i)$:

$$\begin{aligned} \delta_i &= e_i \cdot f'(a_i) \\ &= e_i \cdot a_i \cdot (1 - a_i) \end{aligned}$$

The computation of the deltas of neurons in the output layer $L-1$ can be done in time:

$$2 \cdot n_{L-1} \cdot (t_{sub} + t_{mul}) \tag{4.11}$$

After computing the deltas of neurons in the output layer, each neuron in the previous layer computes its error as the in-product of its output weights and the corresponding delta values of its output neurons (see Figure 4.2).

$$\begin{aligned} e_i &= \sum_{k=0}^{n_l-1} (\delta_k \cdot w_{ik}) \\ \delta_i = e_i \cdot f'(a_i) &= e_i \cdot a_i \cdot (1 - a_i) \end{aligned}$$

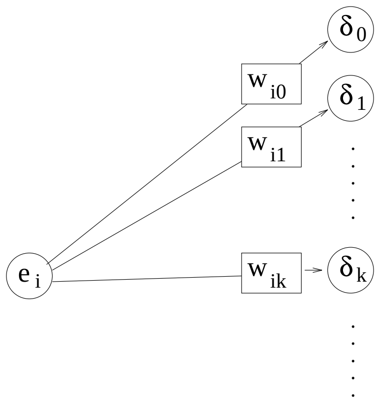


Figure 4.2: *Computation of the hidden errors following $e_i = \sum \delta_k \cdot w_{ik}$, the corresponding deltas are computed as $\delta_i = e_i \cdot f'(a_i)$.*

The "backpropagation" of the errors is performed between each pair of layers l and $l+1$ where $l \in [1 \cdots L-1]$. This involves that no computation of errors and deltas is required for the input layer. The computation of the error and delta values costs time:

$$\sum_{l=2}^{L-1} (n_{l-1} \cdot ((n_l + 2) \cdot t_{mul} + t_{sub})) \tag{4.12}$$

Once the delta values are computed for each neuron, the weight changes are computed as given by Equation (4.13), after which the weights are updated:

$$\begin{aligned} \Delta w_{ij}(t+1) &= \eta \cdot a_i \cdot \delta_i + \alpha \cdot \Delta W_{ij}(t) \\ w_{ij}(t+1) &+= \Delta w_{ij}(t+1) \end{aligned} \tag{4.13}$$

The total time required for the training phase comprises the time for computing the errors and deltas on the output layer (4.11), the time to compute the errors and deltas for each hidden layer (4.12), the time required for computing the weight changes (Equation (4.13), and the time for updating the weights:

$$2 \cdot n_{L-1} \cdot (t_{sub} + t_{mul}) + \sum_{l=2}^{L-1} (n_{l-1} \cdot ((n_l + 2) \cdot t_{mul} + t_{sub})) + w \cdot (2 \cdot t_{add} + 3 \cdot t_{mul}) \quad (4.14)$$

4.2.3 Pitfalls when using arithmetic timings.

The performance of various computer systems and the code made with several compilers with different compiler options was measured for the arithmetics operations `add (+)`, `sub (-)`, `mul (*)` and `div (/)`. The algorithm to measure the timings used three arrays of data a,b,c of varying size (see algorithm 4.4). Though trivial, this code mirrors the usage of arithmetic operations in neural network simulation programs fairly representatively.

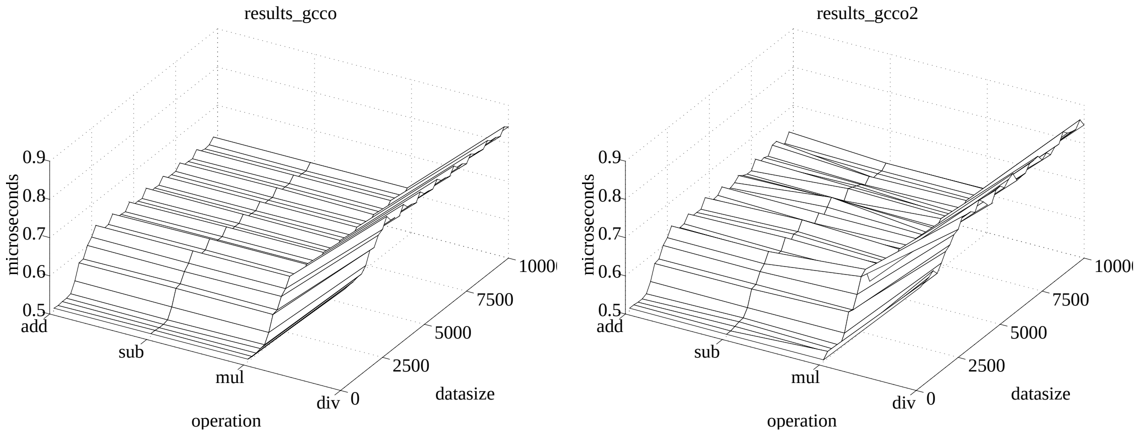
```
float a[datasize],b[datasize],c[datasize];

float time_op()
{
        int i,j,t1,t2;

        t1 = clock();
        for (i=0;i<niterations;i++)
                for (j=0;j<datasize;j++)
                        a[j] = b[j] op c[j];
        t2 = clock();
        return (float) (t2-t1)/datasize/niterations;
}
```

Algorithm 4.4: *Algorithm for measuring the arithmetic performance.*

As explained in section 4.1.5, there exist numerous factors that determine and influence this performance. This can be observed when examining for example Figure 4.3, which depicts the time required for the four arithmetic operations, on a sun sparc station 10 model 30 (32 Mb, 36 MHz) with the gnu gcc compiler using no compiler options or the option -O2. Note that for small data sizes the time for an arithmetic operation is lower than that for larger data sizes. This can be accounted for by the use of internal cache memory. Furthermore note that when using just the plain gcc compiler, the times show a much smoother plot than using the -O2 option.



(a) *Compiled with* `gcc`. (b) *Compiled with* `gcc -O2`.

Figure 4.3: *Time of arithmetic operations* `add`,`sub`,`mul`,`div`*, measured on sparc 10, using algorithm 4.4 compiled with gcc.*

Note that the deviations from some average time for an arithmetic operation are large, especially in Figure 4.3(b). If the basic timings show such large variations, it can be assumed that a performance model that uses them will be very inaccurate. This can be further observed when measuring for example the SOM neural network simulation programs analyzed in the previous section. The time required for activating and training a network of n neurons, N dimensions and p patterns is given by Equations (4.7) and (4.9) respectively. Plotting this expected computation time against the measured time gives the Figure 4.4 depicted below. The time was measured as a full application benchmark. Apparently, the expected times are too high, which can be explained by the opportunities the compiler finds to optimize the code.

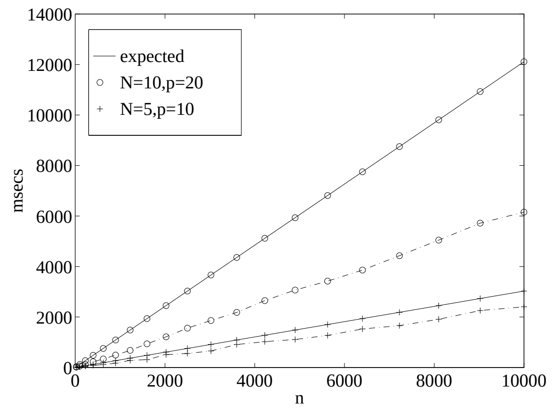


Figure 4.4: *Expected and measured computation times for Kohonen SOM. The solid lines represent the expected times based on the arithmetic timings. The dotted lines represent the*

measured timings. For this figure, the number of neurons is varied and two experiments with $(n, p) = (10, 20), (5, 10)$ *are depicted. Similar results are obtained for other settings of these parameters.*

What can be concluded when using performance modeling on the arithmetic level as described in this section, is that using the timings for individual arithmetic operations results in bad predictions. In the next section, a method is introduced that models the performance of an application in terms of function kernels, which gives much better results.

4.3 Combining Benchmarking and Modeling.

The performance prediction method that is introduced in this thesis tries to eliminate the problems that occur when modeling an application on a too small level of detail. By defining function kernels which represent the salient features of a simulation program, and measuring these for several problem sizes, the expectation is that the corresponding computation times can be extrapolated with higher precision towards the expected computation times for other problem sizes.

4.3.1 Identification of function kernels.

When considering neural network simulation programs, a hierarchy of function kernels can be identified (see Table 4.3), (1) program level, (2) routine level, (3) subroutine level and (4) code fragment level. Though there does not exist a heuristic to choose the proper level on which function kernels have to be identified, by analyzing the code of a simulation program like for example in Sections 4.2.2.1 and 4.2.2.2, the suitable function kernels can be chosen.

program level	measure a complete program
routine level	measure time for routines
subroutine level	measure time for subroutines
code fragment level	identify parts of a subroutine and measure their time

Table 4.3: *Hierarchy of function kernels.*

For a given number of function kernels n_f, the total computation time can be expressed as the sum over the number of times each function is called (N_i), times the time it takes to compute each function (t_i):

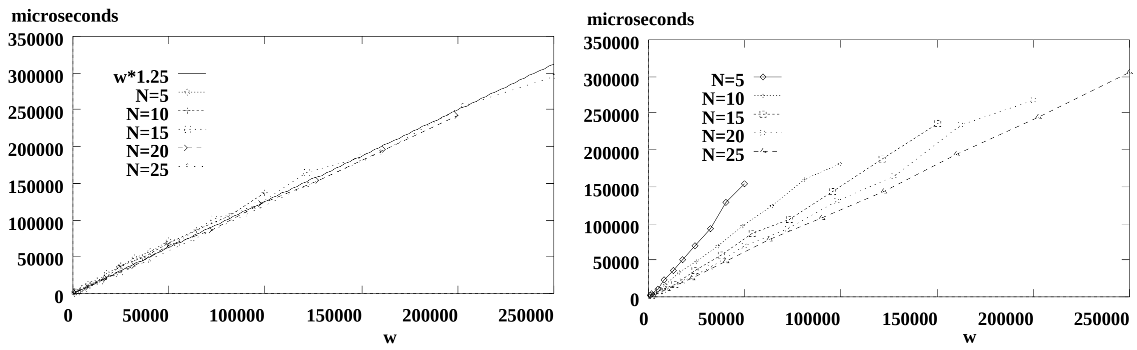
$$t_{calc}(n, w, p) = \sum_{i=1}^{n_f} (N_i(n, w, p) \cdot t_i) \tag{4.15}$$

Function kernels on the program level contain the complete program. Typically, these kernels are used when measuring the performance for a full application. Therefore measuring the performance on this level can be compared to application benchmarking discussed in Section 4.1. The program level is used in this thesis to compare the predicted times for a given application with the times actually measured for the complete program. In Equation (4.15), $n_f = 1$, $N_1 = 1$, and t_1 is the measured overall computation time.

Function kernels on the routine level typically measure the time for routines that are called from within the main body of a program. Consider for example algorithm 4.1. This program contains 4 function kernels, `load_data`, `initialize_network`, `find_winner`, and `update_weights`. For a large number of patterns or large neural networks, the time for loading data and initiating the network can be ignored. The total execution time expressed in routine kernels would then sum up to:

$$p \cdot (w \cdot t_{find\_winner} + n^k \cdot N \cdot t_{update\_weight})$$

Using the same settings of compiler, compiler options and target machine as used in Section 4.2.3, the time for two routine kernels (computing the winner and updating the weights) was measured for different sizes of the Kohonen SOM neural network. The results are depicted in Figures 4.5(a) and 4.5(b).



(a) *Time for* `find_winner`.

(b) *Time for* `update_weights`.

Figure 4.5: *Time for function routine kernels*

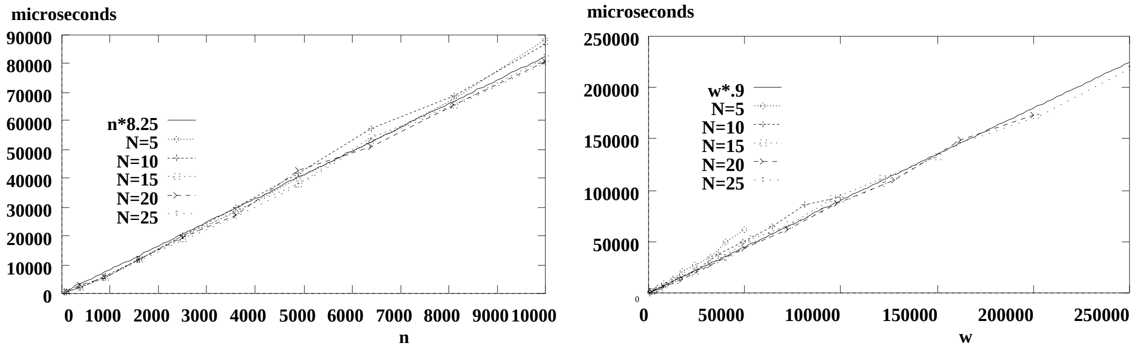
As can be observed in Figure 4.5(a), there exist a linear relationship between the number of weights and the time to find the winner, $w \cdot t_{find\_winner}$, with $t_{find_winner} = 1.25\mu$seconds. However, for Figure 4.5(b), there is more information required, as no such relationship seems to exist. For finding the information, the function kernel at the routine level has to be further subdivided, which can be done by identifying subroutines or program parts that represent significant portions of the execution time. Consider the routine `update_weights`, depicted in algorithm 4.5.

```
#define X(c,i) (abs(c%width-i%width))
#define Y(c,i) (abs(c/height-i/height))
void update_weights (int N, int c, int i, int p)
{
        int dx = X(c,i); int dy = Y(c,i);
        float mult = lrate*gauss[dy][dx];
        float *x,*w;
        int j;
        x = patterns[p];
        w = weights[i];
        for (j=0;j<N;j++,w++,x++)
                *w += mult*(*x - *w);
}
```

Algorithm 4.5: *Algorithm for updating weights of the SOM.*

For each of the n^k neighbors of the winner c, the Gauss value with respect to c has to be 'looked-up', which must be done by computing its x and y offset from c in the Kohonen map. Apparently, measuring only the routine kernel `update_weights` is not sufficient, it should be subdivided into code fragment kernels that measure the computation of x and y offsets and the actual computation of the new weights. The time for updating the weights can thus be modeled as:

$$t_{update\_weights}(n^k, N) \quad = \quad n^k \cdot (t_{offsets} + N \cdot t_{update})$$



(a) *Time for computing x and y offsets.* (b) *Time for updating weights.*

Figure 4.6: *Time for code fragment kernels*

Figures 4.6(a) and 4.6(b) depict the time measured for respectively computing the x and y offsets and updating the weights. From these it can be derived that – using the compiler and machine settings as described above – $t_{offsets} = 8.25\mu$seconds and $t_{update} = 0.9\mu$seconds.

Figure 4.7 depicts the measured time for the program kernel `kohonen.c` and the expected time computed via:

$$\begin{aligned} t_{calc}(n^k, N) &= t_{update\_weights}(n^k, N) + w \cdot t_{find\_winner} \\ &= n^k \cdot (t_{offsets} + N \cdot t_{update}) + w \cdot t_{find\_winner} \end{aligned}$$

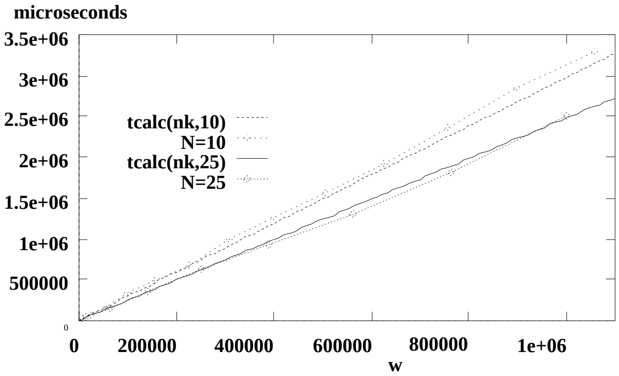


Figure 4.7: *Expected and measured time for the Kohonen SOM, expectations have an accuracy within* $[0 \cdots 5\%]$.

Note that in Figure 4.7, it is possible to predict the calculation time using the time for routine kernels and code fragment kernels with high precision. Furthermore it is interesting to note that using the combination of kernel benchmarking and program modeling, it can be deduced what parts of the program are really compute intensive. In this case the time to compute the x and y offsets causes that the computation time for $N = 10$ is larger than for $N = 25$ for the same number of weights w, as in the latter case the number of neurons is smaller. Based on this observation the decision could be made to allocate a table with 2D-Gaussian coefficients for the complete Kohonen map instead of the $n^k * n^k$ neighborhood, if the available memory resources allow this[3]. The computation of x and y offsets would then be superfluous.

4.3.2 Consequences for parallel programs.

When predicting the execution time of parallel implementations of neural network simulation programs two new problems are introduced, (1) load balancing and (2) communication. In the introduction, the first problem was solved by assuming a perfect load balance. This involves that when using P processors it can be assumed that the calculation time is reduced by a factor P. This assumption can safely be made for decomposition techniques that equally divide a network over the available processors. Especially if the size of a neural

[3]The implementation of the Kohonen SOM within the PREENS algorithm library uses this feature.

network is relatively large compared to the number of processors, in such a case the load will be perfectly balanced. The second problem must be solved by analyzing the patterns of communication in the parallel simulation program. A general approach to analyzing the communication complexity is given in Section 4.2. For any parallel simulation program where it is assumed that the load is well balanced, an upper bound on the communication costs can be given via Equation (4.5) on page 54. However in practice, each individual implementation has to be analyzed in order to arrive at a precise model of the communication costs, instead of such an upper bound.

In the models used in this thesis, a perfect load balance is assumed and the costs for the typical patterns of communication are specified in a precise manner. The consequence of our model for decomposition techniques where the load is not equally balanced involves that the total calculation time can no longer be divided by the number of processors. However, if it is possible to find out how large the largest component to be computed on a processor is, an upper bound can be given for the calculation time (which is the time to compute the largest component).

Another assumption that is made is that the calculation and communication times can be summed up in order to arrive at the total execution time. For parallel programs that implement an overlap between communication and computation, this no longer holds. However, in such a case the model can still be used as an upper bound.

5

A point-to-point communication layer

Outline

In this chapter the design, implementation and performance of a point-to-point communication layer is discussed. The communication layer is suited for MIMD processor systems that communicate via message passing. It is particularly equipped with means to efficiently implement the typical communication paths required for distributed neural network simulations. Before the communication layer is introduced, first an overview over different decomposition techniques is given, pointing out the typical communication requirements for parallel neural network simulations. The implementation and communication costs for *broadcast* and *gather* communications are discussed for all three execution platforms used in this thesis.

5.1 Decomposition techniques

The problem of decomposing a given neural network over a parallel processor system with given topology has often been addressed in the literature. Chu and Wah [18], Cosnard *et al* [20], Witbrock and Zagha [124], and various other authors have discussed the implementation of backpropagation networks on diverse parallel processor systems. Similar efforts have been reported implementing Kohonen networks by for example Obermayer *et al* [74], Wu [125] and Ultsch [104]. Several other neural networks like Hopfield networks [22, 4, 36], EDANN networks [102], ART networks [120] etc. have also been implemented on parallel architectures. In the sequel, when referring to the term *networks*, neural network (simulations) are meant and the term *processor networks* is used when referring to multi-processor systems.

Most of these parallel implementations were *ad hoc*, the PNNS were specifically decomposed based on careful examinations of both the neural network architecture and the characteristics and configuration of the target platforms involved. In general, using this way of implementation gives good results. A number of efforts have been made to implement tools for automatically mapping a given neural network architecture onto a given processor system [18, 101]. Although the mapping problem has been acknowledged to be an NP-hard problem, the resulting mappings can give reasonable efficiencies and the tools support a user with a mechanism via which there is no need to be occupied with parallel programming. On the other hand, in most cases the resulting implementations are not as efficient as would be possible with dedicated decompositions. Furthermore it has appeared that, at least for certain classes of neural networks, implementing them is relatively easy to do. In [38, 100, 110], several methods for decomposing neural networks on MIMD multi-processor systems are discussed. They can be distinguished in three levels of decomposition, *job-level* decomposition, *dataset* decomposition and what we call *network* decomposition. The first two techniques are coarse grained decomposition methods which are well suited for MIMD parallel computer architectures, as the job sizes are large and the amount of inter-processor communications is small. The latter technique is defined as the collection of decomposition methods that can be used to divide a given neural network onto a given processor system. In general, these techniques are more fine grained than the former ones and thus require more inter-processor communications.

5.1.1 Job-level decomposition

Job-level decomposition is a coarse grained parallelization method which places complete copies of a neural network (the jobs) on different processors. In general, each job is initiated with different parameter or architectural settings. The technique is often used by researchers looking for the proper initial values for which a neural network is able to perform well for a particular application. They just start up a number of copies of the neural network program on different machines (workstations) and evaluate their performance based on convergence and generalization criteria. In [76], this method has evolved in a tool which automatically selects a proper architecture and set of learning parameters for the back-propagation neural network. Each processor runs a complete copy of the neural network, together with an evaluator process which extracts information for generating evaluation statistics from the neural network simulation. Based on the evaluation of a neural network residing on a specific processor, the decision can be made to quit the evaluation and use that processor for evaluating a neural network started with another set of initial values. Note that for applications which require very large networks or large sets of data, this method cannot be used directly as it is restricted by the amount of memory available per processor. On the other hand, by combining this approach with other techniques in which each job is decomposed over a number of processors, larger sized networks can be handled. In this thesis, job level decomposition will not be discussed. Instead, the attention will be focussed on the other two techniques.

5.1.2 Dataset decomposition

Dataset decomposition is a special kind of job-level decomposition. Each job is initiated with the same parameters and architecture, so the same neural network is present on every processor. The parallelism that is exploited in this technique stems from the dataset. If we consider a neural network as operating in two phases, a *training* and *recall* phase, for both an efficient decomposition technique can be exploited by dividing the dataset in a number of parts and computing each part in parallel. For the recall phase, the state of activation A_i is computed based on the network (weight) state $W(t)$ and the input pattern x_i:

$$A_i(t) \quad = \quad recall(W(t), x_i)$$

The computation of all activations for all p input patterns can be performed via:

$$A_{(0 \cdots p-1)}(t) \quad = \quad \bigcup_{i=0}^{p-1} A_i(t)$$

This can be done in parallel on P processors via 5.1:

$$A_{(0 \cdots p-1)}(t) \quad = \quad \bigcup_{proc=0}^{P-1} \quad \bigcup_{i=proc \cdot p/P}^{((proc+1) \cdot p/P)-1} \quad A_{(proc,i)}(t) \tag{5.1}$$

Similarly for the training phase, the change of a weight variable of a network is computed based on its current weight state and a new training pattern i as:

$$\Delta W_i(t) \quad = \quad train(W(t), x_i)$$

If somehow a neural network is able to be trained using *epoch* or *batch* learning, this involves that the weight changes for all patterns can be computed by summing up the individual weight changes per pattern as:

$$\Delta W_{(0 \cdots p-1)}(t) \quad = \quad \sum_{i=0}^{p-1} \Delta W_i(t)$$

This can be performed in parallel via:

$$\Delta W_{(0\cdots p-1)}(t) \;=\; \sum_{proc=0}^{P} \; \sum_{i=proc\cdot p/P}^{((proc+1)\cdot p/P)-1} \Delta W_{(proc,i)}(t) \tag{5.2}$$

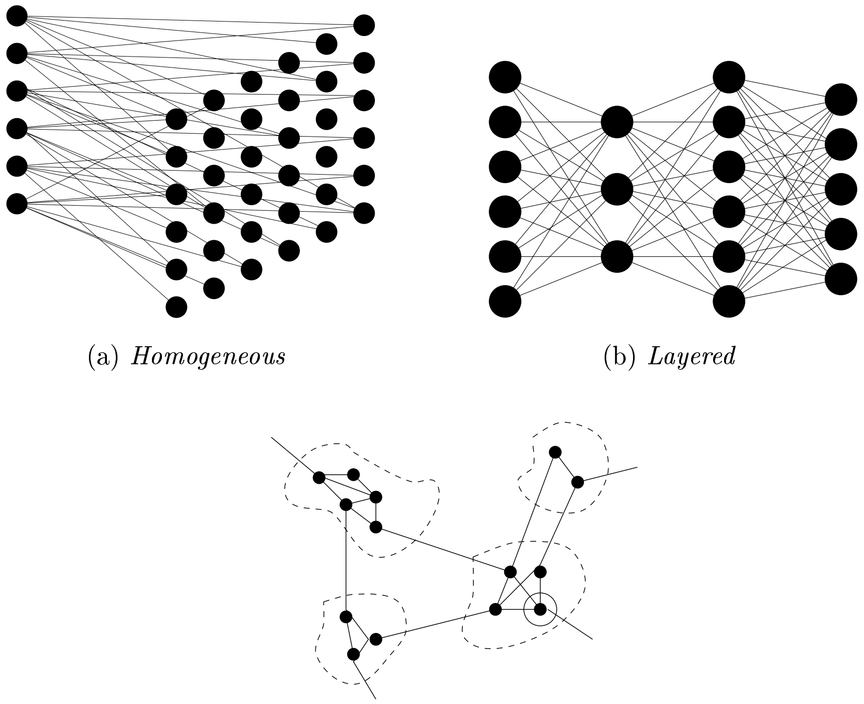
Especially if the datasets are large, dataset decomposition is a highly efficient parallelization technique as during the calculation of the sub-terms in 5.1 and 5.2, no communication is necessary.

5.1.3 Neural network decomposition

Job-level and dataset decomposition result in coarse grained (and thus efficient) decompositions of the problem domain. However, they cannot be exploited if the neural networks are too large to fit on one processor. Allocating one job per processor is only feasible if the processor's resources are sufficient. For jobs representing neural networks which are too large to fit on one processor, each job has to be divided into a number of components running on several processors. The collection of parallelization methods via which this can be performed, is called network decomposition. In order to efficiently use a processor network for speeding up a specific problem by dividing it over the available processors, two items are of major importance: load balance and synchronization and communication [6, 37, 48]. A proper load balance ensures that each processor in the network has an equal amount of work to do. If this would not be the case, some processors would still be working while others are ready to proceed but have no job to do. By dividing the network in components with equal work loads, the load can be balanced properly. Considering the architecture of a neural network and the load of its individual components, several ways of decomposing it into a number of parts exist, each on a certain level of parallelism or grain size. Apart from job-level and dataset parallelism as discussed above, two more levels can be distinguished.

The first is on the level of the neurons and their connections. It seems a straightforward way to decompose neural networks by considering the network architecture and performing a one-to-one mapping of neurons onto processors and connections onto processor links. Some processor architectures have been proposed where neural networks are implemented in hardware or on special purpose neurocomputers or coprocessors that function as neural accelerator boards [41, 43, 79, 98]. Furthermore, fine grained implementations have been reported of Kohonen [75] and backpropagation networks [95, 127] on the Connection Machine. However, as this thesis is concerned with coarse or medium grained general purpose machines where the number of processors is much smaller than the number of neurons in the neural network, this level of parallelism will not be discussed in great detail. Rather than looking at the fine grained parallelism as observed on the level of neuron operations, larger jobs have to be identified like the ones described in the previous sections.

The second kind of parallelism is on the level of clusters of neurons and their connections. Decomposing and mapping neural networks on this level of parallelization has been discussed in for example [38, 94, 100, 126]. Based on the connectivity and load of the different clusters, the decision can be made to group them into one component and place this on a processor. Considering their architecture and learning mechanisms, four classes of neural networks can be distinguished, which are described below. Each has specific features that require different decomposition strategies in order to maintain a good load balance while reducing synchronization and communication times. The four classes are homogeneous networks, layered networks, heterogeneous networks and dynamic networks (see Figure 5.1).



(a) *Homogeneous* (b) *Layered*

(c) *Heterogeneous*

Figure 5.1: *Neural network architectures, dynamic networks can have any architecture, where one or more nodes "grow" during a training phase.*

Homogeneous networks

In case of homogeneous networks such as the Hopfield or Kohonen networks, the network can be divided into a number of components via geometric decomposition [100]. Each part is running as one separate task on a processor. If several subtasks would run on the same processor, unnecessary overhead regarding to scheduling and intertask communications would be introduced. As will be pointed out in Chapter 7, for certain neural networks geometric decomposition results in a bad load balance. Especially when different parts of a network have different computational loads, this is the case. A technique that can be used to avoid this problem is scattered decomposition [100], which distributes parts of the network randomly over the available processors.

Layered networks

When layered networks like the backpropagation model are used, each layer can be subdivided over all processors. Each processor has thus control over a subpart of each layer. Note that in this case all-to-all communications are required as every two subsequent layers are fully interconnected. In [94], it is discussed that the different layers can be placed on a number of processors, using a special case of pipelined parallelism as discussed in [121]. In Chapter 7, this technique will be further explained.

Heterogeneous networks

If modular, heterogeneous networks are considered, each module should be placed on one processor. This is because in general each module consists of densely interconnected clusters of neurons, whereas the connectivity between different modules is low. A general method to decompose these networks is the following. If a module is too large to fit on one processor, or if two modules are connected, a number of adjacent processors should be used where the number of inter-processor communications should be minimized [38].

Dynamic networks

In some cases, neural networks have a dynamic architecture, i.e. they grow during the training process. Examples of these networks are the (recurrent) cascade-correlation neural networks [33, 32], CALM networks [71], ART networks [13, 12, 14, 15] or the grownet algorithm [96]. These networks impose a completely different requirement on the decomposition problem, i.e. dynamic load balancing. If at a certain state during training the load is well balanced, growing one or more neural network components could result in a poorly balanced load. To solve this problem, the load has to be somehow dynamically re-balanced during the training phase.

5.2 Synchronization and communication

Synchronization and communication problems are typical for distributed neural network simulations. Each component calculates its new state, after which it has to exchange this information with the components it is connected to. The communication overheads are determined by the communication resources of the target execution platform and the kind, size and number of required communications.

5.2.1 Communication networks

For each parallel processor system where data has to be communicated between processors, some kind of communication network is required. Some architectures devise special, highly optimized communication routines that can be exploited, like described by Zhang [127] for the Connection Machine. Other architectures like the DAP or MPP have high speed communication mechanisms that shift data across rows or columns of the processor arrays. For shared memory multi-processor systems, data is communicated via a data medium (bus) that is shared by the different processors. For MIMD processors like transputer systems or the Intel hypercube, the basic communication primitives are point to point data exchange between neighboring processors that are connected to each other via some communication link.

Typically, these systems consist of a set of processing elements (PE) and a communication network which connects the processing elements with each other. For transputer systems, each processing element has 4 communication links via which it can connect to other PE. The implementations discussed in this thesis all run on transputer systems configured in a grid or tree topology.

5.2.2 Communication requirements

Each of the parallel neural network simulations discussed in this thesis, is implemented via a master process and several slave processes. The master process performs the I/O with the user-interface and file system. It also sends commands and distributes the data to the slave processes. Each slave process hosts a part of the neural network, or part of the data. They communicate with each other and with the master process. For parallel neural network simulations, a master-slave implementation operates in two phases, a calculation phase and a synchronization phase. During calculation, each process operates independently of the others. After each calculation step, processes enter the synchronization phase during which information is communicated. In [110, 111], the typical patterns of communication that are needed for parallel neural network implementations on MIMD computers are distinguished. *Broadcast* communications are needed for making information globally available to every slave process. In cases where distributed information has to be somehow accumulated, *gathering* communications are needed that collect the information from slave processes to the master.

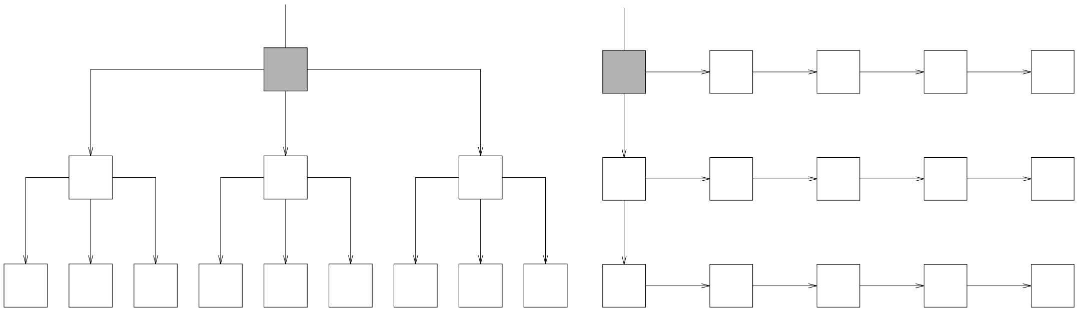
Broadcasting

For master-slave implementations, typical broadcast communications contain data that has to be available on every processor, or commands that rule a program's flow of control. Algorithm 5.1 illustrates a characteristic main loop of a slave program:

```
while (broadcast_command (&command)) {
  switch (command) {
          ......
  }
}
```

Algorithm 5.1: *Typical slave main loop.*

Depending on the topology of a processor network, various implementations of a broadcast operation can exist. However, in all cases, broadcasting data is performed from the master process to the slave processes. The master process typically resides on the root (host) processor of the processor network. In Figures 5.2(a) and 5.2(b) it is depicted how broadcasting data on a tree and grid topology is performed.



(a) *Broadcasting on a tree (depth=2).* (b) *Broadcasting on a 5x3 grid.*

Figure 5.2: *Broadcasting messages on tree and grid topologies. The marked nodes are the root processors, which host the master process.*

From Figure 5.2(a) it can be deduced that broadcasting data can be done in parallel on every node at the same depth of the tree. For a depth of 1, the master sends the data to each of its sons. As each processor node in a transputer tree has maximum three links available to connect to son processors, this requires at most 3 communications. For each depth greater than 1, the processes residing on processors on this depth can receive the data from their father processor and send it further to their sons. This can be done in parallel for each processor. The total times for broadcasting data of size s on a processor tree of P processors can be estimated via (5.3), where the depth of a tree of P processors can be estimated by $^3log\ P$.

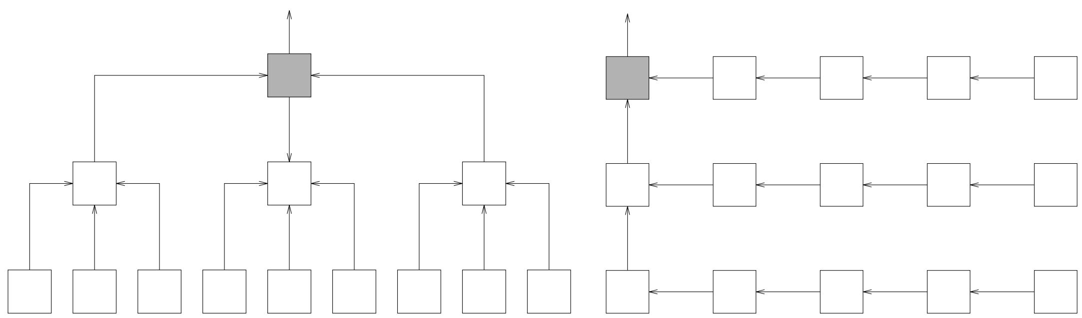
$$T^{tree\_broadcast}(P,s) \quad = \quad 3 \cdot depth(P) \cdot s \cdot t_{comm} \tag{5.3}$$

Considering Figure 5.2(b), broadcasting data is done by the master process by sending it to its right and down neighbors. Furthermore, each processor in the leftmost column receives the data from his top processor and sends it to its right and down neighbors. Each processor which does not reside in the leftmost column receives the data from its left neighbor and transmits it further to its right neighbor. For each row, this can be done in parallel. For a $W \times H$ grid, $H - 1$ communications are required for sending the data downwards. Furthermore, $W - 1$ communications are required for sending data from left to right in each row. The total times for broadcasting data of size s on a processor grid of P processors can thus be estimated as:

$$T^{grid\_broadcast}(P, s) = (width(P) + height(P) - 2) \cdot s \cdot t_{comm} \quad (5.4)$$

Gathering, accumulating and collecting

Gathering information in transputer grid or tree topologies is performed from the leaves of the processor network up to the root processor. The amount of inter-processor communications required for performing gathering operations is thus the same as given in Equations 5.3 and 5.4. One important issue with gathering is to try to minimize the number of subsequent inter-processor communications. Furthermore, if possible, it should be tried to perform any required computations on the data during the gathering phase, as this means that the computations can be performed in parallel.



(a) *Gathering on a tree.* (b) *Gathering on a grid.*

Figure 5.3: *Gathering weight changes on tree and grid topologies*

When considering Equation (5.2), the weight changes do not only have to be gathered, but also have to be summed up. This can be done in parallel on each node as illustrated in Figure 5.3. Each node receives the weight changes from its neighbors, accumulates these with its own weight changes and sends the accumulated sum to its father process. Note that using this technique, not only the required communications are performed in parallel, but also the needed accumulations. For a neural network containing w connections, the total times for gathering and accumulating the weight changes can be estimated as below, where t_{acc} represents the time for accumulating two weight values.

$$T^{tree\_gather\_accumulate}(P, w) = 3 \cdot depth(P) \cdot w \cdot (t_{comm} + t_{acc}) \quad (5.5)$$

$$T^{grid\_gather\_accumulate}(P, w) = (width(P) + height(P) - 2) \cdot w \cdot (t_{comm} + t_{acc}) \quad (5.6)$$

For gathering the required information in case of network decomposition, in most cases such simple gathering strategies as described above do not suffice. For dataset decomposition, the only communications required are the gathering and accumulation of weight changes. As their data have identical sizes on each process, the administration involved is rather simple. On the other hand, for network decomposition in general each process manages a different part of the neural network components and therefore, their corresponding data are not identical and could even have different sizes in case the load is not uniformly distributed. The latter is mostly the case as the load can only be distributed equally over the processors if the size of all neural network components is divisible by the number of processors. For example for a backpropagation neural network containing n_l neurons in one of its layers, the load is only uniformly distributed if $n_l \bmod P$ equals zero.

One solution to solve this problem is to use virtual point-to-point connections, where each slave process sends its data directly to the master. Most operating systems, like Helios, Parix or CS-tools offer this possibility. Other (runtime) systems like ParC require user-developed routers that perform this task. However, it will appear in the subsequent chapters that using this solution introduces too much subsequent inter-processor communications. Rather than using virtual point-to-point communications, it is more efficient to communicate locally. However, solutions using this method require that slave processes know about the data that they are going to communicate with neighboring slaves. Some administration of the distribution of data is needed that has to be present on every slave process. In Chapter 7, such an implementation is discussed.

Based on the typical patterns of communication observed in neural network simulations, a set of broadcast and gather operations was designed and implemented, specifically tuned for parallel neural network applications running on tree and grid topologies. On the GCel and PX only grid topologies are used. This is because no physical tree topology could be installed as the processors are arranged in a grid. Furthermore, mapping virtual tree topologies on a grid resulted in unpredictable and inefficient communication performance, as will be discussed below.

5.3 Communication paths in trees and grids

Three kinds of communication routines are contained in the communication layer: 1) routines for setting up the communication on a grid or tree topology, 2) routines for broadcasting information and 3) routines for gathering information. The latter two routines are introduced in the next section. The routine `query_grid()` finds out the dimensions of the physical grid topology and arranges for each process the communication channels `NORTH`, `EAST`, `SOUTH` and `WEST`. The routine `query_tree()` finds out the physical tree topology and arranges for each process the communication channels `TOP`, `LEFT`, `DOWN` and `RIGHT`. *Broadcast routines* send messages from a master processor (root) to all other processors. The master process sends a message to each neighboring slave process. Each subsequent slave process transmits the message to its neighbors further up the path. *Gather routines* receive information from all processors to a master processor. Each slave process receives information from its neighbors further up the path, does some processing on the information and transmits it in the direction of the master.

5.3.1 Setting up the communication in a grid

As explained in the previous chapter, the way in which communication channels are set up when using Helios or Parix differs significantly. However, for the applications used in this thesis the following assumptions are made:

1. Each master process runs on processor (0,0) in the $W * H$ grid.
2. Each slave process i runs on processor $((i+1) \bmod W, (i+1)/W)$ in the grid. Slave processes are numbered from `[0 ... nslaves-1]`.
3. Each process on the left-most column of the grid is connected to its `NORTH`, `SOUTH` and `EAST` neighbor.
4. The other processes are connected to their `EAST` and `WEST` neighbors.

After setting up the communication channels, a number of global variables are set via which a process can find out how many neighbors it has, whether it has neighbors to its east or south side, what communication channels are available, etc. In particular, a process' identification `slave_id` is set to either -1 indicating that the process is the master, or a number i within `[0 ... nslaves-1]`. Figure 5.4 depicts the communication channels that are set up, how the master and slave processes are identified via their identification and how the processes are partitioned on a physical grid.

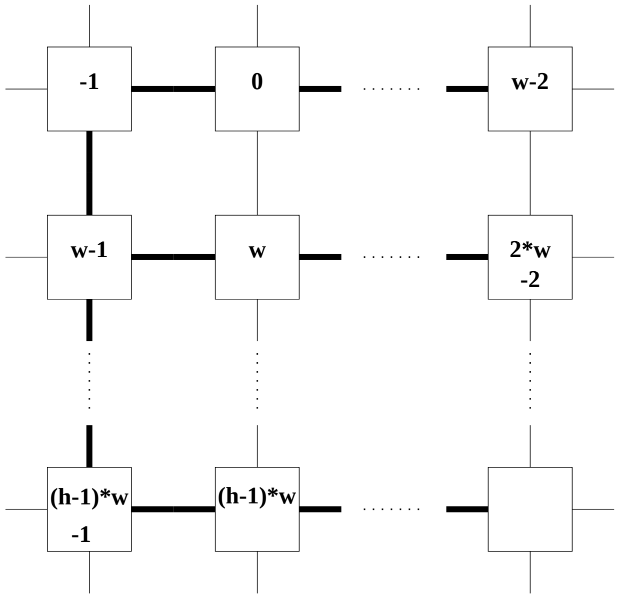


Figure 5.4: *Situation after calling* `query_grid()`.

The routine `query_grid()` gets as parameters the width and height of the taskforce grid (i.e. the tasks that are assigned to processors in the grid). In its main routine, each program calls `query_grid()` to check whether the required taskforce matches the physical topology, and if so, to set up the communication channels and initialize the variables used for broadcast, gather and point-to-point communications.

5.3.1.1 Setting up the communication in Helios

The Helios Resource Management Library [23] contains a number of routines that can be used to investigate the physical processor network. Using the routine `RmGetNetwork()`, a datastructure representing the status of the transputer network is filled in. In `query_grid()`, this structure is examined for the number of processors and connections between them. For each processor, its identification can be requested via `RmGetProcessorID()`. Based on the id, its position in the grid can be determined and the required identification of its neighbors can be computed. For each of the directions in which the processor must be connected, it is checked whether it is connected to the required processor. This is performed via the routine `RmFollowLink()`, which returns the identification of the processor that is connected via a specific link. If the physical configuration matches the topology required by a grid, the identification of the communication channels (file descriptors) is computed. Otherwise an error message is returned. The assignment of file descriptors to a process' specific communication channels in its `NORTH`, `EAST`, `SOUTH` and `WEST` directions is done as depicted in Figure 5.5.

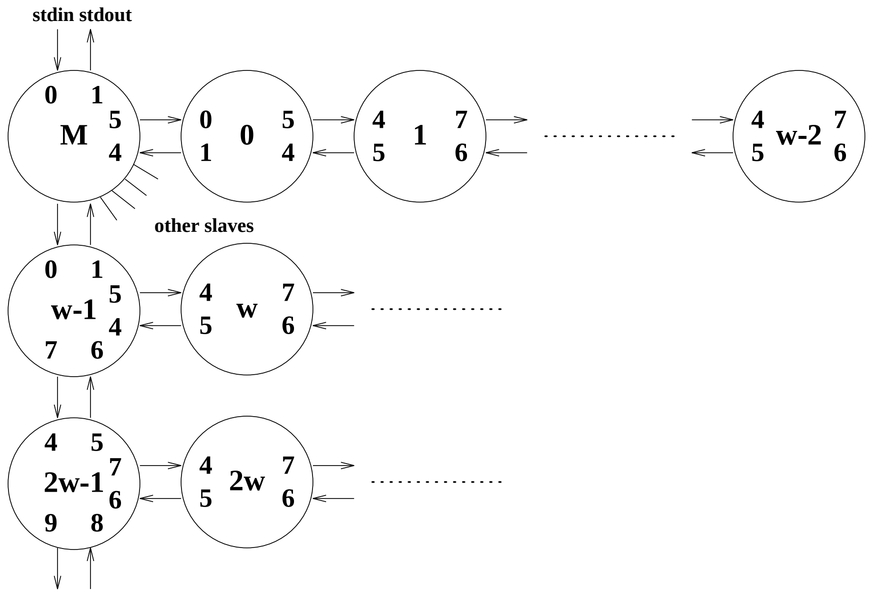


Figure 5.5: *Assigned posix file descriptors after calling* `query_grid()`.

Note that the master is connected via virtual channels to all slaves. The channels are identified via the following file descriptors:

```
#define from_slave(i)  (4+i+i)
#define to_slave(i)    (5+i+i)
```

The `SOUTH` channels of the master are connected to slave `w-1`, and each pair of channels is connected to the file descriptors 0 and 1 of each corresponding slave. Based on its `slave_id`, the routine `query_grid()` can determine which file descriptors must be valid for a certain task. Using the Helios routine `fdstream()`, it can be checked whether the corresponding channel is valid or not. If all required file descriptors are valid and the physical configuration is correct, this means that the loader of the CDL script has correctly loaded each task on the corresponding processor and that all communication channels are setup. Subsequently, `query_grid()` returns `true` and an application may be started that uses neighboring point-to-point communications between tasks running on neighboring processors.

5.3.1.2 Setting up the communication in Parix

In order to write portable code, the communication layer has to be independent of the application that runs on it. Therefore, the Parix implementation of `query_grid()` has the same semantics as the Helios implementation described above. As discussed before, one of the main differences between Parix and Helios is that with the latter system, a user has to claim a resource map in order to allocate a transputer network, after which he must use the CDL mechanism to specify and partition a task force. In Parix, this login procedure is not necessary. Instead, a program is loaded onto a transputer system directly from the host operating system. The same program is loaded onto each processor and therefore it has to be decided at runtime which `slave_id` a task has. As we use a one-to-one mapping of tasks onto processors, using the Parix call `GET_ROOT()`, which returns a local and global description of the processor network, it can be determined which processor the task runs on. If this processor is not part of the `W*H` grid, the task exits, as depicted in the code below:

```
procID = GET_ROOT()->ProcRoot->MyProcID;
x = GET_ROOT()->ProcRoot->MyX;
y = GET_ROOT()->ProcRoot->MyY;
slave_id = y * W + x - 1;
if (x>=W||y>=H) exit (0);
```

The communication channels are dynamically created via `ConnectLink()`, which allocates a channel to a specified processor. Note that when using Helios, it is checked whether the physical topology is correct and whether the communication channels are all validly allocated via the CDL mechanism. Using Parix, a previously installed partition of processors is allocated. If this succeeds, it is guaranteed that the partitions are configured in a certain way (i.e. grid). Allocating partitions and loading tasks is done via the Parix `run` command:

```
run -ppartition task_code [task_arguments]
```

The check whether the communication channels are valid is implicitly performed when using `ConnectLink()`. If each `ConnectLink()` succeeds, the corresponding channels are valid and `query_grid()` returns `true`.

5.3.2 Setting up the communication in a tree

As mentioned above, mapping a virtual tree topology onto a physical grid does not make much sense. Experiments with using Parix within the CAMPP'93 programme [111] showed that the communication performance suffers significantly from the through-routing of messages. Furthermore, two other factors determine the suitability of a virtual tree topology. The first is the mapping algorithm that is used. We have used the standard `MakeTree` utility from Parix, which tries to create a proper virtual tree topology on a given physical grid. The other is the size of the grid onto which the tree is mapped. One would expect that the larger the grid onto which a tree is mapped, the better the mapping result. However, consider for example the timings of gather-accumulate-broadcast operations of a virtual tree of 40 processors mapped onto three different physical grids:

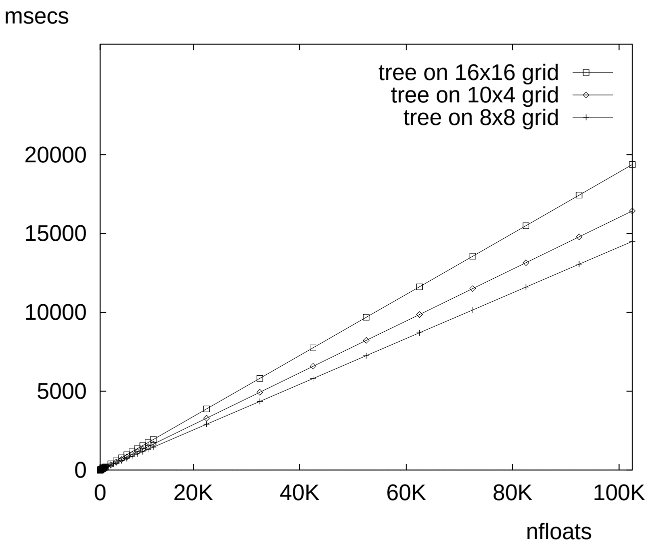


Figure 5.6: *Timings of GAB on a 40-ternary virtual tree.*

As can be expected, the timings differ. But it also appears that the times for a 16x16 grid are considerably larger than for smaller grids, which is not the expectation. Based on these observations, no further experiments with Parix and virtual trees on the GCel are performed. On the NSC, physical tree configurations can be specified via a resource map. Similar to the way in which tasks set up the communication channels in a grid configuration using `query_grid()`, each task explicitly checks on which processor it is running. Note that in Helios a programmer has to specify the resource map and CDL-script. For tree topologies, some utilities were designed and implemented to support a programmer with these tasks. The utilities use a simple configuration algorithm to specify a tree topology. If this algorithm is used, for each processor (or process), the following hold:

1. Each processor is connected to a father processor via its `TOP` link, the master processor has no father.
2. Each processor is father of one to three subtrees where the number of processors contained in a subtree $N(subtree) \geq 0$ (a processor with no subtrees is a leave).
3. If a processor has one or more subtrees, it is connected via its `LEFT`, `DOWN`, or `RIGHT` link.
4. The difference between all $N(subtree)$ on a certain depth of the tree never exceeds 1, i.e. the processor tree is balanced.

The same algorithm is used to specify a CDL task force description, which implies that a direct mapping of tasks onto processors is established. When running a task force, each task must call the routine `query_tree()`, which finds out whether the processor on which the task runs is physically connected to the proper processors in the tree. This routine also uses the tree configuration algorithm, so the same algorithm is used on three levels, 1) to create a resource map, 2) to create the corresponding CDL task force, 3) to assure that each task runs on the appropriate processor. The latter is done similar to `query_grid()` by using the Helios routines contained in the Helios library `RmLib`. Furthermore, it is checked by using `fdstream()`, if each task has valid file descriptors in its `TOP`, `LEFT`, `DOWN` and `RIGHT` link. After calling `query_tree()`, the communication channels are assigned to posix file descriptors as depicted in Figure 5.7.

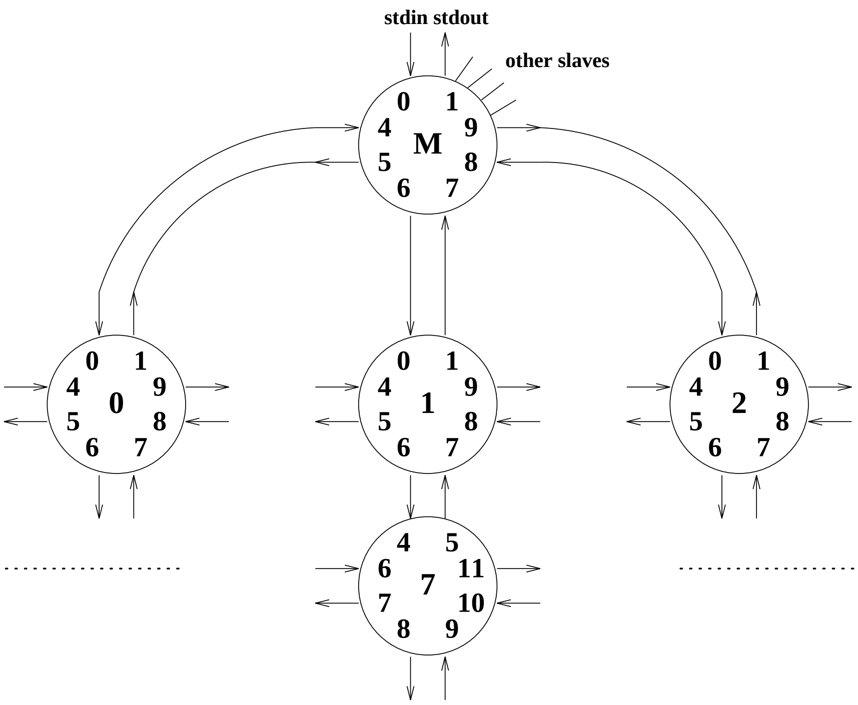


Figure 5.7: *Assigned posix file descriptors after calling* `query_tree()`.

5.3.3 Communicating in a tree and grid

Once the communication channels are setup correctly, neighboring tasks can communicate to each other using the channels. In Helios, this is performed via posix `read()` and `write()` calls, whereas when using Parix, this is done via the routine `RecvLink()` and `SendLink()`. Each of these routines has three important parameters, the direction to or from which to communicate, the memory location where the data is located, and the number of bytes the data occupies. By using the following declarations and macros, the same application code can run on both Helios and Parix (where `l` represents the link number [0–3], `b` represents the address where the data is located and `n` represents the number of bytes:

```
#ifdef parix
extern LinkCB_t *links[4];
#define rec(l,b,n) RecvLink(links[l],b,n)
#define snd(l,b,n) SendLink(links[l],b,n)
#endif
#ifdef helios
extern int links[4];
#define rec(l,b,n) read(links[l],b,n)
#define snd(l,b,n) write(links[l]+1,b,n)
#endif
```

Using the routines `query_grid()` and `query_tree()`, on each processor a number of variables via which it can find out to which neighbors is has to communicate are set. For both a tree and a grid, these are called `has_left`, `has_top`, `has_right` and `has_down`. Using these variables and the macros described above, point-to-point communications can be used between neighboring processors.

Unfortunately, it is impossible to write applications using the grid or tree point-to-point neighboring communication schemes in a transparent way independent of the underlying topology. If processes on a grid have to communicate to each other, the have to talk to their respective `NORTH`, `EAST`, `SOUTH` and `WEST` neighbors. If the processes lay on a tree, they have to know whether to talk to their `TOP`, `LEFT`, `DOWN` and `RIGHT` neighbors. However, for many applications some typical communication patterns are required. By examining the patterns of communication, a software layer can be designed on top of the point-to-point layer, that implements these for e.g. a grid or tree topology. For the parallel neural network simulations described in this thesis, such a software layer was made. The layer contains communication routines for one-to-all broadcasts and all-to-one gather operations.

5.4 Broadcast and gather routines

Broadcast and gather routines differ in the direction of the flow of information in a processor network. Using broadcasts, information is distributed over the processor network from one processor to all other processors. Using gathering, information is collected from all processors to one processor. With broadcasting for each processor the data to receive from and to transmit to neighboring processors is the same. This means that the amount and type of data is equal for all processors. With gathering, in general the data to collect is distributed over the processor network. This means that depending on the distribution of the data, the amount and even the type of the data to be communicated may differ for distinct processors. Depending on which processor a process is running on, and what neighbors it is connected to, it has to decide via which links to receive or send messages. Several possibilities to realize this exist:

1. The first is to compile different programs for each of the cases, and make sure to load the proper program on the corresponding processor. For example, this may involve that a program `master` is loaded on the root processor, a program `slave_column` on the left most column, and a program `slave_row` on each row of a grid.

2. The second is to assign the proper routines to the general communication routines during the examination of the network topology. For example this can be realized for broadcasts by using a function pointer that represents a `row_broadcast`, `col_broadcast`, or `root_broadcast`.

3. The third option is to determine the receipts and sends dynamically by examining some global variables set by the topology querying routines `query_grid()` and `query_tree()`.

The first option is rejected because this requires a large number of different programs and a lot of extra administration for task distribution and execution. The second option could be used, but can only be exploited for general routines like broadcast and gather-accumulate operations. However, parallel neural network applications decomposed via network decomposition (see Chapter 7) also require more application specific communication schemes. This means that the global variables mentioned above have to be used anyway, and as furthermore the performance gain compared to the third option is minimal (a small number of comparisons are made), option three has been chosen.

5.4.1 Broadcasting

The variables that are examined when broadcasting in a grid are `has_left`, `has_top`, `has_right` and `has_down`. Algorithms 5.2(a) and 5.2(b) depict a simplified version (shows no error detection/reports) of the code used for broadcasting on grids and trees.

```
int broadcast (char *buf, int size)
{
  if (!has_left) {
    if (has_top)
      rec (NORTH,buf,size);
    if (has_down)
      snd (SOUTH,buf,size);
  }
  else
    rec (WEST,buf,size);
  if (has_right)
    snd (EAST,buf,size);
}
```
(a) *Broadcast on grids.*

```
int broadcast (char *buf, int size)
{
  if (has_top)
    rec(TOP,buf,size);
  if (has_left)
    snd(LEFT,buf,size);
  if (has_down)
    snd(DOWN,buf,size);
  if (has_right)
    snd(RIGHT,buf,size);
}
```
(b) *Broadcast on trees.*

Algorithm 5.2: *Broadcasts on grids and trees.*

5.4.2 Gathering

For many applications where the decomposition of a program is of a homogeneous nature, gathering data becomes relatively easy to accomplish. Examples are the accumulation of vectors or the computation of a global maximum or minimum value. Both are used in the applications discussed in this thesis. The algorithm to gather and accumulate a number of vectors of equal length is depicted in algorithm 5.3.

```
int f_gather (float *dst, float *buf, int n)
{
  int i, size = n*sizeof(float);
  if (has_right) {
    rec(EAST,(char *)buf,size);
    for (i=0;i<n;i++)
      dst[i] += buf[i];
  }
  if (!has_left) {
    if (has_down) {
      rec(SOUTH,(char *)buf,size);
      for (i=0;i<n;i++)
        dst[i] += buf[i];
    }
    if (has_top)
      snd(NORTH,(char *)dst,size);
  }
  else
    snd(WEST,(char *)dst,size);
}
```
(a) `GA()` *on grids.*

```
int f_gather (float *dst, float *buf, int n)
{
  int i, size = n*sizeof(float));
  if (has_left) {
    rec(LEFT,(char *)buf,size);
    for (i=0;i<n);i++)
      dst[i] += buf[i];
  }
  if (has_down) {
    rec(DOWN,(char *)buf,size);
    for (i=0;i<n);i++)
      dst[i] += buf[i];
  }
  if (has_right) {
    rec(RIGHT,(char *)buf,size);
    for (i=0;i<n);i++)
      dst[i] += buf[i];
  }
  if (has_top)
    snd(TOP,(char *)dst,size);
}
```
(b) `GA()` *on trees.*

Algorithm 5.3: *Gathering and accumulation on grids and trees.*

5.5 Gather, accumulate, and broadcast

As mentioned before, all-to-all broadcasts are implemented using a subsequent gathering and broadcasting of information. In Chapter 7, all-to-all broadcasts for non-uniform data are discussed. In this section it is assumed that the data to be gathered is equal on every processor and that during the gathering, all data elements have to be summed up. This is exactly the case when using dataset decomposition, where on each processor the weight changes are computed, after which they have to be accumulated and made available to every other processor. The routine that subsequently gathers, accumulates and broadcasts information is called `GAB()`. A number of experiments were carried out to quantitatively validate the times required for performing `GAB()` on diverse tree and grid topologies.

5.5.1 Setup for the experiments

The time to perform `GAB()` for a vector of size s is (respectively for a grid and tree):

$$T^{\mathtt{GAB}\_grid}(P,s) = (width(P) + height(P) - 2) \cdot s \cdot (2 \cdot t_{comm} + t_{acc}) \tag{5.7}$$

$$T^{\mathtt{GAB}\_tree}(P,s) = 3 \cdot depth(P) \cdot s \cdot (2 \cdot t_{comm} + t_{acc}) \tag{5.8}$$

When considering Equations (5.7) and (5.8), two function kernels required to quantitatively model the times for `GAB()` can be identified. The first is the well-known ping_pong kernel, which measures the time t_{comm} for transmitting a value between two neighboring processors over one physical communication link. This benchmark starts a timer, transmits a message over a link, and subsequently receives the same message and halts the timer. The communication time is the difference between the two timers divided by two. The second kernel to be benchmarked is the accumulation kernel, which measures the time to accumulate two vectors. As mentioned in Chapter 4, many factors may determine the performance of such a kernel. Therefore, the kernel we used for measuring the accumulation times was contained in a larger piece of code mimicking some typical neural network simulation program. The next table lists the results for the ping_pong and accumulation kernels:

Machine	t_{acc}	t_{comm}
NSC	6.00 μseconds/float	3.00 μseconds/float
GCEL	4.99 μseconds/float	3.59 μseconds/float
PX	0.31 μseconds/float	3.79 μseconds/float

Table 5.1: *Times for ping_pong and accumulation kernels.*

Using these timings for t_{comm} and t_{acc}, the expected times for Equations (5.7) and (5.8) can be computed. The next sections present the measured and expected times for `GAB()`, measured on different processor topologies and network sizes on the Nijmegen Super Cluster (NSC), the GCel-512 (GCEL) and the PowerXPlorer (PX). For each of the experiments, vectors of size [10 ... 90], [100 ... 900], [1000 ... 9000] and [10000 ... 100000] floats where used in order to examine the model.

5.5.2 `GAB()` on the NSC

One of the main problems with the Helios operating system is that it is not suited for very large processor systems. Especially if the number of communication channels is large, Helios often crashes. If for example all-to-all communications between processes would be implemented using virtual communication channels, where each process receives and sends its information directly to all other processes, it has appeared that this is not feasible using Helios (version 1.22). Applications running more than 36 fully connected tasks just died. Using the local communication harness described in the previous chapter, applications can be scaled up to the full 64 processors of the NSC. Furthermore, this solution has the advantage that, — especially for `GAB()` —, local communications can be performed in parallel (communication as well as accumulation).

`GAB()` on NSC grids

Figure 5.8 depicts the measured and expected times for `GAB()` on NSC grids.

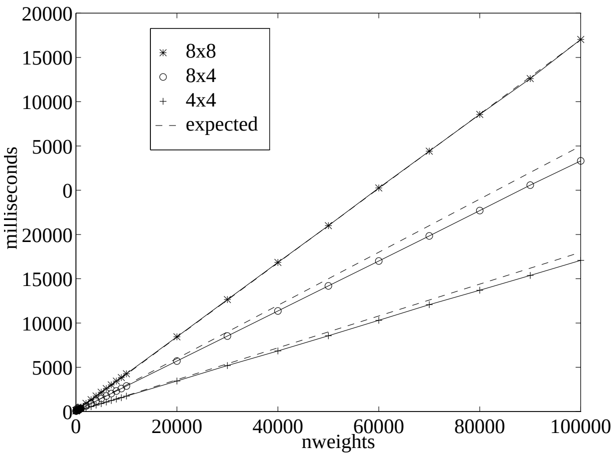
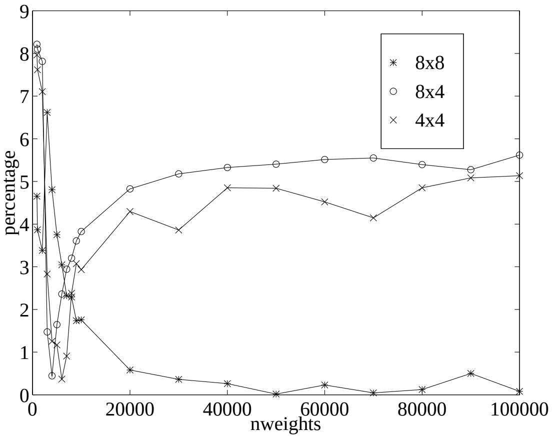
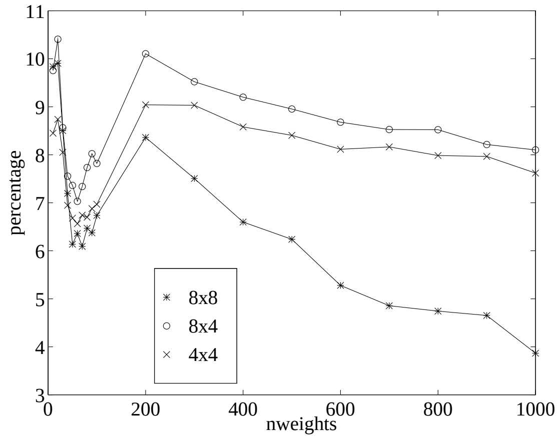


Figure 5.8: *Expected and measured times for* `GAB()` *on NSC grids.*

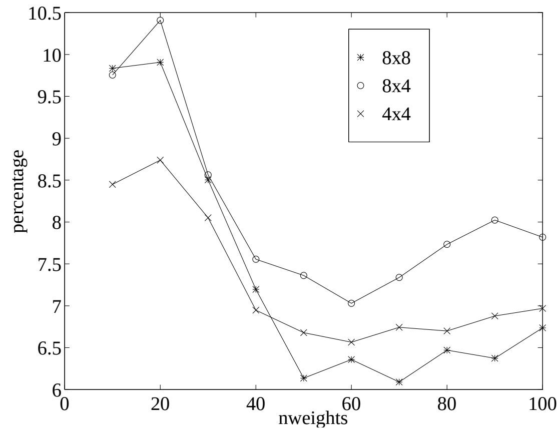
The expectations are modeled following Equation (5.7), where for t_{comm} and t_{acc} the times depicted in Table 5.1 are filled in. It can be observed that the expected times are modeled relatively good for large messages (<6%), whereas for small messages the deviations become much higher (up to 10%). This behavior is depicted in Figures 5.9(a) to 5.9(c).



(a) *Large sizes (1000 … 100000).*



(b) *Medium sizes (0 … 1000).*



(c) *Small sizes (0 … 100).*

Figure 5.9: *Deviations between measured and expected times (in %).*

The routine `GAB()` is used in parallel neural network simulation programs to accumulate a number of weight vectors. Recall that our goal is to find out the suitability of transputer systems for a neural network simulation by predicting the performance (and thus execution times) for a given simulation on a given machine. This is done via modeling the calculation and communication times. According to the results presented above, relatively good predictions on NSC grids can be made for neural networks containing more than 400 weights. For a backpropagation network, this means an architecture of say 10 inputs, 20 hidden and 10 output neurons. For a Kohonen network an architecture of 10 inputs and 5x8 neurons contains 400 weights. Both are moderately sized networks. For really small networks, from these results it can be concluded that the predictions will probably be not too precise, but on the other hand, if a relatively large number of communications is required, transputer systems are only suited for relatively large neural networks. The suitability of transputer systems for parallel neural network simulations will be discussed in the subsequent chapters.

`GAB()` on NSC trees

The expected times for `GAB()` on a ternary tree are given by Equation (5.8). For each depth of the tree, 3 subsequent communications and accumulations are required. Note that for trees that are not completely filled, this can be considered as an upper bound for the number required. For example for a tree containing 64 processors, the first 40 nodes form a completely filled tree of depth 3, where 24 of the 27 nodes laying on depth 3 are connected to a leave node. This means that instead of 3 subsequent communications and accumulations between depth 3 and 4, only 1 is executed. In Figure 5.10, the expected and measured times are depicted on trees of 13, 40 and 64 nodes. Note the difference between the dashed line above $tree_{64}$ and the dash dotted line exp_{64}. The first is the expected time for 64 nodes following (5.8), whereas the second is the expectation where only one communication and accumulation is counted between depth 3 and 4.

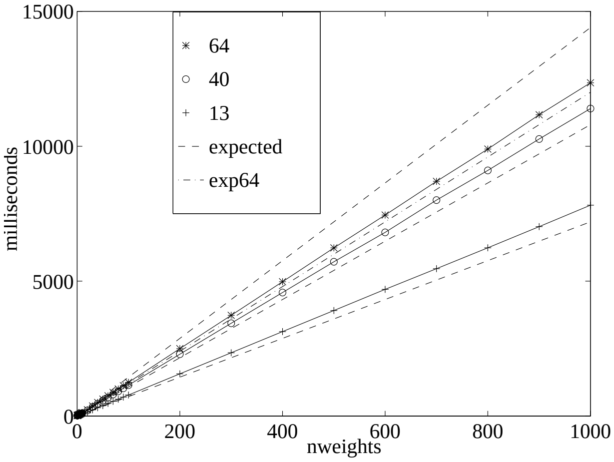
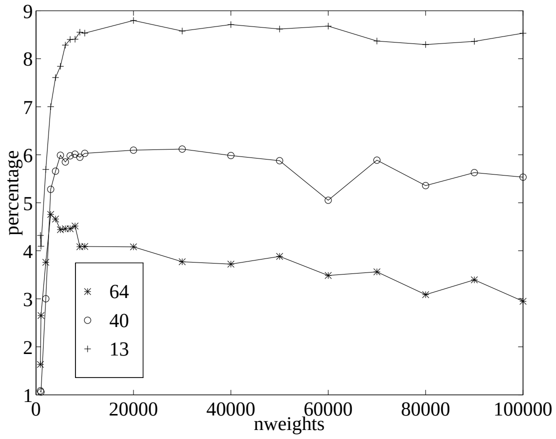


Figure 5.10: *Expected and measured times for* `GAB()` *on NSC trees.*

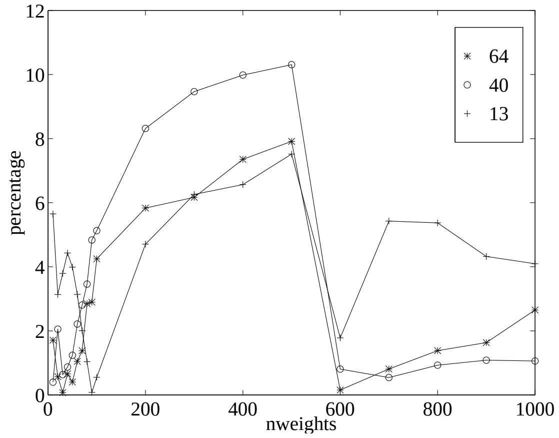
Note that indeed, the expectations for 64 nodes following (5.8) are too high, whereas exp_{64} provides an accurate expectation, which is represented by:

$$t_{exp64}(s) \quad = \quad T^{\texttt{GAB\_}tree}(40, s) + s \cdot (2 \cdot t_{comm} + t_{acc})$$

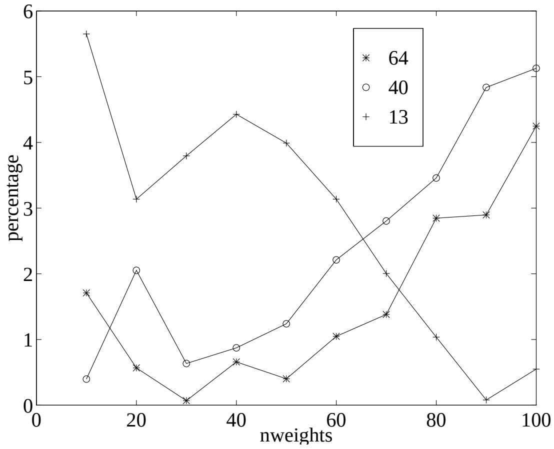
The next figures depict the deviations for large, medium and small message sizes.



(a) *Large sizes (1000 ... 100000).*



(b) *Medium sizes (0 ... 1000).*



(c) *Small sizes (0 ... 100).*

Figure 5.11: *Deviations between measured and expected times (in %).*

It shows that, similar to the results for NSC grids, for relatively small messages the deviations can be as large as 10%. There can be several reasons for the deviations, all of which are caused by the simplicity of the model. For example issues like the setup time, the amount of internal versus external memory that is used, the number of crossbar switches that are traversed during communication, or *synchronization failures*, i.e. some processes are ready while others are still computing, are not captured in the model. The latter issue can be illustrated as follows. Assume some process on a processor is gathering information from its neighbors (i.e. its sons in a tree, or its `EAST` or `DOWN` neighbors in a grid). Following the model described here, for each processor it is assumed that at a specific time t all data to be gathered is available and can be sent to its neighbor. However, if for some reason the data is available at a later moment $t + \delta$, it can occur that the actual time after which gathering has finished is higher than expected. Another effect may cause the measured time to be smaller than the expected time. Consider for example a node a as depicted in Figure 5.12.

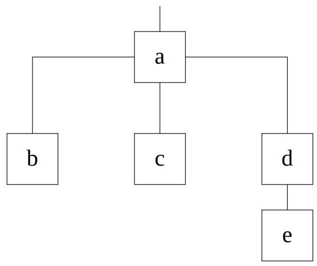
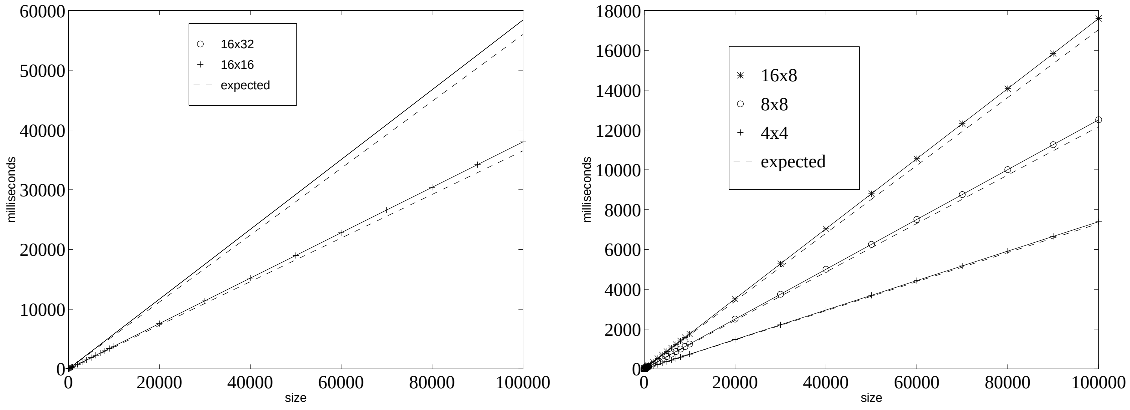


Figure 5.12: *Situation in which the expected times are lower than the measured times.*

In this situation, the model predicts the time for `GAB()` as $6 \cdot s \cdot (2 \cdot t_{comm} + t_{acc})$, which is modeled more precisely as $exp_5 = 4 \cdot s \cdot (2 \cdot t_{comm} + t_{acc})$. However, when considering algorithm 5.3(b), node a first gathers the data from its `LEFT` and `DOWN` sons, after gathering it from its `RIGHT` son. In the situation of Figure 5.12, this means that exp_5 is not even good enough, the expected time should be $3 \cdot s \cdot (2 \cdot t_{comm} + t_{acc})$, because during the gathering of data from processors b and c, processor d has already gathered its data from e. on the other hand, the expected time following our model will still give an upper-bound on the execution time, which can be used to assure that at most a certain maximum time is required for `GAB()`.

5.5.3 `GAB()` on the GCel-512

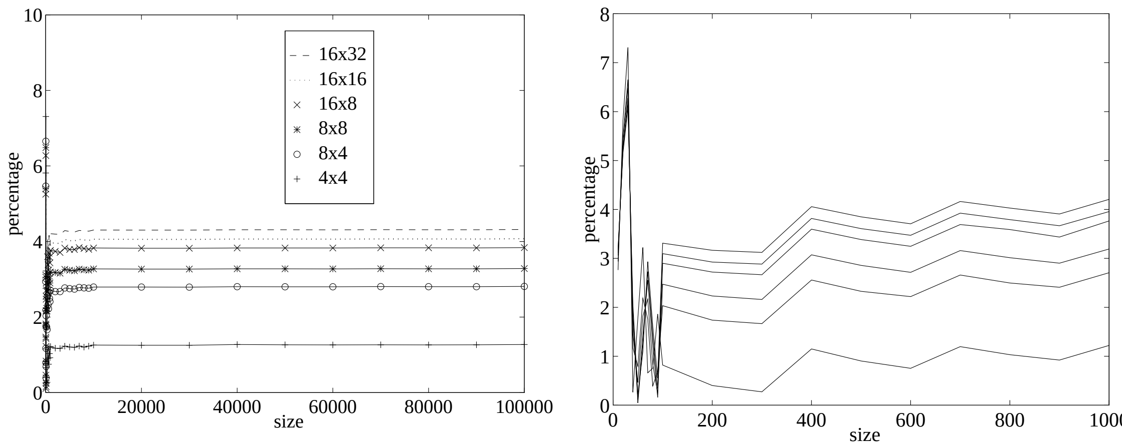
The same set of experiments was carried out on the GCel machine, which enabled the validation of the model for significantly larger grid topologies. The program to measure `GAB()` was ported to Parix, and t_{comm} and t_{acc} where measured as 3.59 and 4.99 μseconds per float respectively. Note that a node on the NSC is a T800 transputer running at 25 MHz, and a node on the GCel is a T805 running at 30 MHz. Surprisingly, the naive expectation that the accumulation time would thus be a factor $^{25}/_{30}$ lower is true! The difference in communication times is not caused by a difference in link speed, but can be effected by a difference in the implementation at the system level of communication, or in the difference between the exploited crossbar switches. Figures 5.13(a) and 5.13(b) depict the measured and expected times for `GAB()` on GCel grids of topology 4x4 to 16x32.



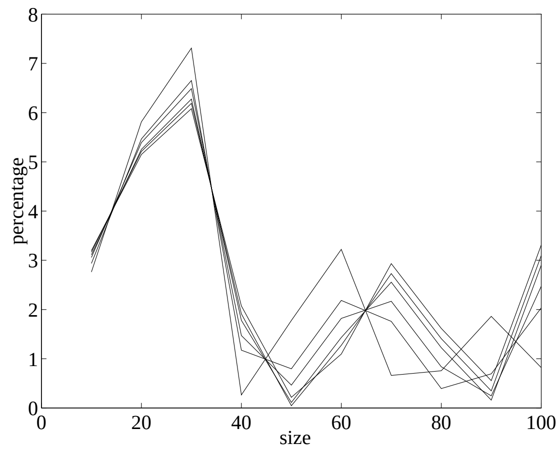
(a) *For 16x16 and 16x32 grids.* (b) *For 4x4, 8x4 and 8x16 grids.*

Figure 5.13: *Expected and measured times for* `GAB()` *on different GCel grids.*

Again, it appears that the model is able to predict the measured times accurately. This can also be observed by considering Figures 5.14(a), 5.14(b) and 5.14(c) which depict the deviations for large, medium and small message sizes.



(a) *Large sizes (1000 ... 100000).* (b) *Medium sizes (0 ... 1000).*



(c) *Small sizes (0 ... 100).*

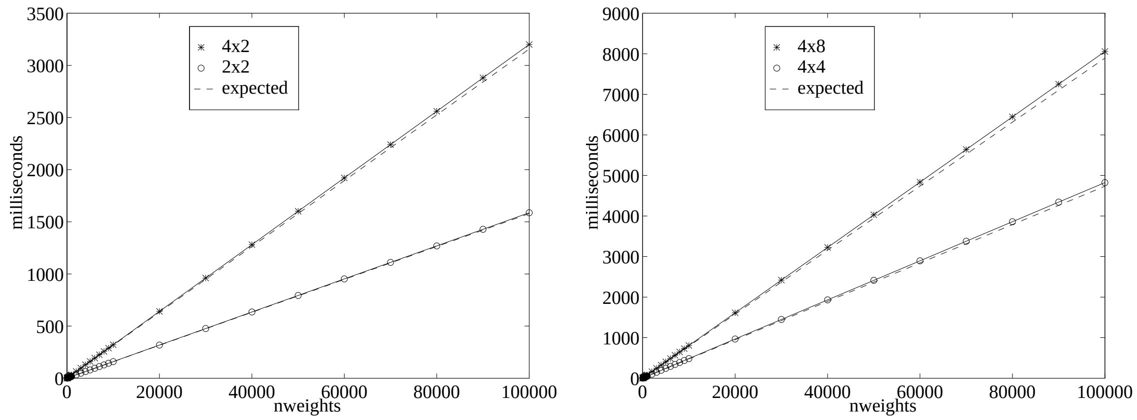
Figure 5.14: *Deviations between measured and expected times (in %).*

For relatively large vectors, the deviations are within 4.3%. What is noted, is that for small messages the deviations are not very high either, opposed to the situation with the NSC.

5.5.4 `GAB()` on the PowerXPlorer

A node on the PowerXPlorer (PX) is a PowerPC with a peak performance of 80 MFLOPS. The T805 offers 4.3 MFLOPS, and – again naively –, we may expect the accumulation time to be $4.3 * 4.99/80 \approx 0.27$. The measured t_{acc} amounts to .31 μseconds/float, which is surprisingly close to the expected time. The communication time t_{comm} is 3.79 μseconds/float. This involves that the ping-pong times for the PX are comparable to that of the GCel.

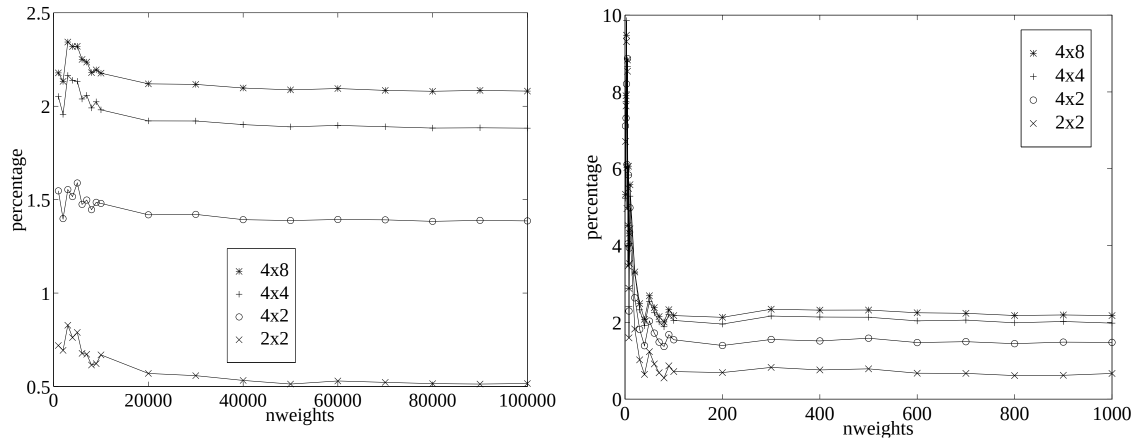
Consider that the underlying communication network is implemented through a transputer system, and that for each inter-processor communication an extra data transfer is required through the shared memory of the PowerPC and communication transputer. In first instance one would expect that the communication performance would suffer from this overhead. However, the implementation of shared memory transfers is very fast and furthermore, each transputer can be fully dedicated to communication only. Therefore, each transputer is equipped with an efficient communication micro kernel, where no overheads exist of scheduling and calculation. The result is that one node provides a high computational power and a communication performance comparable to that of the GCel and NSC. In Figures 5.15(a) and 5.15(b) the expected and measured times for `GAB()` on a 2x2, 4x2, 4x4 and 8x4 grid are depicted.



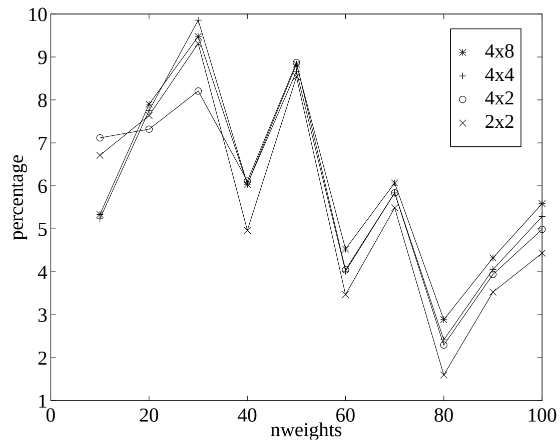
(a) *For 2x2 and 4x2 grids.* (b) *For 4x4 and 8x4 grids.*

Figure 5.15: *Expected and measured times for* `GAB()` *on different PX grids.*

The deviations are depicted in Figures 5.16(a), 5.16(b) and 5.16(c).



(a) *Large sizes (1000 ... 100000).* (b) *Medium sizes (0 ... 1000).*



(c) *Small sizes (0 ... 100).*

Figure 5.16: *Deviations between measured and expected times (in %).*

5.6 Conclusions

Broadcast and gather communication routines have been introduced in this chapter for tree and grid topologies. Following the communication paths that messages pursue using these routines, a model was defined that predicts the communication times for small and large messages. For all three systems (PX, GCEL and NSC) and for NSC trees, the model was validated quantitatively. This was done by measuring the communication times for various sizes of the processor network and a large range of neural network sizes. The timings predicted by the communication model were estimated accurately with a precision of on the average 5%. All predictions had an accuracy of ±10%.

With these communication routines, a foundation is made for implementing parallel applications. In the next chapter, the communication layer defined here will be used to implement two neural network paradigms, the backpropagation and Kohonen SOM neural network.

6

Dataset Decomposition

Outline

Using the combined performance prediction model introduced in the previous chapters, the performance, speedup, efficiency and scalability of a large number of backpropagation and Kohonen neural networks are determined. Kernel benchmarks are identified and measured on the Nijmegen Super Cluster (NSC), and on the GCel-512 (GCel) and PowerXPlorer (PX) located at the University of Amsterdam. The predicted performance measures are subsequently compared to full program kernels, i.e., actual parallel implementations of the two neural networks. It will appear that the combined method can be used to predict the performance for any neural network size on any size of processor network rather accurate, based on kernel benchmarks measured on one processor, and communication kernels measured between only two processors.

6.1 A general dataset decomposition algorithm

When using dataset decomposition for exploiting parallelism in neural network simulations, a set of patterns is equally distributed among a number of identical processes. After this distribution, all processes compute the results for their local patterns in parallel. Subsequently, the results for all processes are gathered and broadcast. This series of actions may be iteratively repeated. Note that like described in the previous sections, a subsequent gather and broadcast communication is performed to emulate an all-to-all broadcast. A general algorithm for dataset decomposition is depicted below:

```
load_data();
initialize_network();
distribute_patterns();
while (!ready) {
  for (p=0;p<nlocal_patterns;p++) {
    compute_pattern(p);
  }
  gather_results();
  broadcast_results();
}
```

Algorithm 6.1: *General algorithm for dataset decomposition.*

In case of neural networks, two phases can be identified in which dataset decomposition can be exploited, a *recall* phase and a *training* phase. In the recall phase, for each local pattern the state of activation of a neural network is computed, after which all computed activations are gathered and broadcast. In the training phase, for each pattern the computed network status is used to compute the change of neural network variables (e.g. weights, biases, learning rate). After all network changes are computed, they are gathered, accumulated, broadcast and used to update each copy of the neural network.

```
while (!ready) {
  for (p=0;p<nlocal_patterns;p++) {
    compute_net_status(p);
  }
  gather_net_status();
  broadcast_net_status();
}
```

(a) *Recall phase.*

```
while (!ready) {
  for (p=0;p<nlocal_patterns;p++) {
    compute_net_status(p);
    compute_net_changes(p);
  }
  GAB_net_changes();
  update_net_changes();
}
```

(b) *Training phase.*

Algorithm 6.2: *General algorithms for recall and training phase.*

This code runs on each processor in the transputer network. For each neural network, the computation of its network status, its network changes and the administration of its changes is different. Furthermore, these computations are independent of the underlying communication mechanism. The gathering, accumulation and broadcasting of the network changes is very similar for different neural network models, but does depend on the communication primitives and underlying processor topology used. Therefore it will appear in the next sections that the difference in the performance model between dataset decomposed backpropagation and Kohonen network mainly shows in the calculation model. This is in contrast to other decomposition methods (e.g. network decomposition), where the patterns of communications differ significantly for different neural networks.

In Equation (2.3), the overall calculation and communication time for a parallel application is expressed as:

$$\begin{aligned} T_{calc}(P,n,w,p) &= \frac{1}{P}\cdot\sum_i (N_i(n,w,p)\cdot t_i) \\ T_{comm}(P,n,w,p) &= C(P,n,w,p)\cdot t_{comm} \end{aligned} \tag{6.1}$$

To find out the calculation time for a dataset decomposed neural network, the $N_i(n,w,p)$ and corresponding time t_i have to be found. This has to be done by carefully examining the code of the simulation program and by identifying and measuring a properly selected set of function kernels. How this can be done is described in Chapter 4, Section 4.3. To find out the overall communication time, the number of required communications for a given processor network with P processors and a given neural network application (n,w,p) has to be determined. Similar to modeling the calculation time, a set of communication kernels has to be identified to determine the number of communications $C(P,n,w,p)$. This amount can be expressed as the sum of the required communications for a number of communication kernels:

$$C(P,n,w,p) = \sum_i (N_i^c(P,n,w,p)$$

The function and communication kernels on the routine level in Algorithms 6.2(a) and 6.2(b) are listed below:

`compute_net_status()`	`gather_net_status()`
`compute_net_changes()`	`broadcast_net_status()`
`update_net_changes()`	`GAB_net_changes()`

Table 6.1: *Function and communication kernels for dataset decomposition.*

If all patterns are equally divided over the available processors, the number of patterns each processor has to compute is `nlocal_patterns`$=p/P$. Using this perfect load balance property, the total calculation time for training one epoch can be expressed as:

$$T_{calc\_train}(P,n,w,p) = \frac{p}{P}\cdot(N_{cns}(n,w)\cdot t_{cns} + N_{cnc}(n,w)\cdot t_{cnc} + N_{unc}(n,w)\cdot t_{unc})$$

where the indices cns, etcetera stand for `compute_net_status()` etcetera. The communication time is the time required for gathering, accumulating and broadcasting the network change information.

The next sections describe the implementation, performance model, and predicted and measured performance for the training phase of dataset decomposed backpropagation and Kohonen neural networks with several architectures. The recall phase is not discussed, because (see Algorithm 6.2(a)) this phase also incorporates $N_{cns}(n, w)$ and t_{cns}. In Section 6.5 and 6.6 the simulations are discussed with respect to the achieved speedup, scalability and efficiency.

6.2 Backpropagation dataset decomposition

When implementing a backpropagation network using dataset decomposition, in fact multiple copies of a sequential algorithm are used to compute the local network status and network changes. Therefore, a similar analysis of the required function kernels can be exploited as described for the sequential Algorithm 4.3. Note that in this algorithm, the weight changes are directly updated instead of gathered, accumulated, and broadcast before the updating takes place. Algorithm 6.3 depicts the dataset decomposition version of Algorithm 4.3.

```
for (epoch=0;error>errcrit&&epoch<nepochs;epoch++) {
  error = 0.0;
  for (pattern=0;pattern<npatterns;pattern++) {
    ComputeActivations(pattern);  /* compute_net_status()    */
    ComputeErrors();              /* compute_net_changes()   */
    ComputeWeds();                /*          ...            */
    error += ComputeError();      /*          ...            */
  }
  GABWeightsAndBiases();          /* GAB_net_changes()       */
  error /= npatterns;             /* update_net_changes()    */
  ChangeWeightsAndBiases();       /*          ...            */
}
```

Algorithm 6.3: *Algorithm for backprop dataset decomposition.*

When analyzing this algorithm, the following function and communication kernels can be identified on the routine level:

kernel	time required
`ComputeActivations()`	t_{act}
`ComputeErrors()`	t_{errs}
`ComputeError()`	t_{err}
`ComputeWeds()`	t_{wed}
`ChangeWeightsAndBiases()`	t_{chg}
`GABWeightsAndBiases()`	$T^{\mathtt{GAB}}(P, nweights + nbiases)$

Table 6.2: *Function and communication kernels for backpropagation neural networks decomposed via dataset decomposition.*

Note that the first four routines are computed for each pattern, whereas the gathering and updating of the weights and bias changes only happens for each epoch. Let T_{pat} and T_{epo} represent the time required per pattern and per epoch:

$$\begin{aligned} T_{pat}(n, w) &= t_{act}(n, w) + t_{errs}(n, w) + t_{err}(n, w) + t_{wed}(n, w) \\ T_{epo}(P, n, w) &= t_{chg}(n, w) + T^{\mathtt{GAB}}(P, n + w) \end{aligned}$$

The total time for computing p patterns on P processors would than amount to:

$$T_{backprop}(P, n, w, p) = \frac{p}{P} \cdot T_{pat}(n, w) + T_{epo}(P, n, w)$$

which — for a large number of patterns — can be estimated via two constants t_{pat} and t_{chg} as:

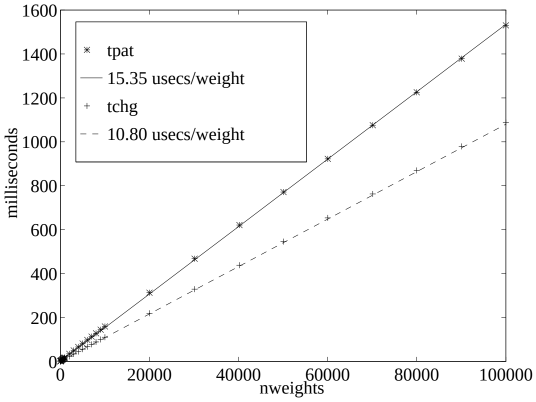
$$T_{backprop}(P, n, w, p) = \frac{p \cdot w}{P} \cdot t_{pat} + w \cdot t_{chg} + T^{\mathtt{GAB}}(P, n + w)$$

During this time, on each of the P processors, for each of the p/P patterns per processor, w weights are computed, so the performance can be expressed in MCUPS as:

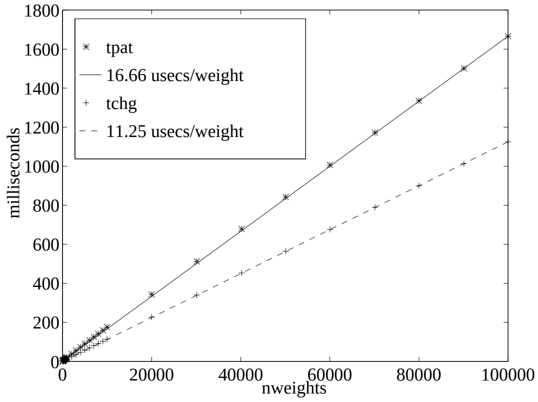
$$\mathcal{P}_{backprop}(P, n, w, p) = \frac{p \cdot w}{T_{backprop}(P, n, w, p)} \tag{6.2}$$

6.2.1 Measurements for function kernels

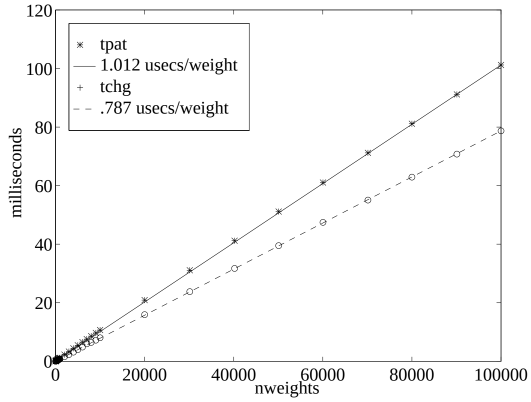
For each of the machines on which the experiments were carried out, the same programs were used. Because the routines used in the underlying communication layer have the same syntax and semantics for both Helios and Parix, porting the code to any of the NSC, GCel or PX machines introduced no difficulties. The results for the measured times t_{pat} and t_{chg} are depicted in Figures 6.1(a), 6.1(b) and 6.1(c).



(a) *Kernels on NSC.*



(b) *Kernels on GCel.*



(c) *Kernels on PX.*

Figure 6.1: *Measured and fitted times for function kernels t_{pat} and t_{chg} in milliseconds.*

Note that the measured data points fit the plotted lines. This means that there exists a linear relationship between the number of weights and t_{pat} and t_{chg}. From Figure 6.1 and Equation (6.2), the following expressions can be derived:

machine	$T_{backprop}(P, n, w, p) = w \cdot (p/P \cdot t_{pat} + t_{chg}) + T^{\texttt{GAB}}(P, n+w)$
NSC	$w \cdot (p/P \cdot 15.35 + 10.80) + T^{\texttt{GAB}}(P, n+w)\mu$seconds
GCel	$w \cdot (p/P \cdot 16.66 + 11.25) + T^{\texttt{GAB}}(P, n+w)\mu$seconds
PX	$w \cdot (p/P \cdot 1.012 + .0787) + T^{\texttt{GAB}}(P, n+w)\mu$seconds

Table 6.3: *Expected time for backprop dataset decomposition.*

6.2.2 A problem with small neural networks

Using this model to predict the performance for backpropagation neural networks decomposed via dataset decomposition results in good predictions as will be discussed in the next sections. However, in first instance the deviations between measured and predicted results for small neural networks were very high. These could amount to more than a factor 4 for networks of 10 weights (a 2x2x3 network). The reason for these large deviations is that the times for function kernels computed for small networks differ significantly from the fitted times. When "zooming in" the plotted times in Figure 6.1 at an interval of [0 ... 1K], it appears that the smaller the neural network, the higher the time to compute t_{pat}. This can be observed in Table 6.4.

nweights	**NSC**	**GCel**	**PX**
10	55	73	4
50	31	39	2.3
100	25	31	1.8
500	19	73	1.3
1K	17	73	1.2
5K	15.8	17.1	1.1
10K	15.4	16.7	1.0

Table 6.4: *Measured time for t_{pat} in μsecs/weight.*

From this table it can be deduced that only for networks >5K weights the function kernels for t_{pat} show the predicted behavior following Table 6.3. The reason for this problem is that each of the times t_{act}, t_{err} and t_{wed} not only depends on the number of weights, but also on the number of neurons. For large networks, the effect can be ignored and the execution time can be modeled via Table 6.3. However for small networks, also the number of neurons has to be taken into account. There are two ways to solve this problem. The first is to use a more detailed model, i.e. to identify and measure kernels on the level of code fragments (see Chapter 4, Section 4.3.1). Unfortunately when considering the code of backpropagation on this level, many individual kernels can be identified, which is hard to examine. Furthermore, it was discussed in Chapter 4 that a too low level of detail can result in bad predictions. The second method that can be used is to fit t_{pat} as:

$$t_{pat} = w \cdot t_{weight} + n \cdot t_{neuron}$$

Using Matlab [66], t_{pat} was fit on the measured function kernels (which were already measured to determine the results from Table 6.3). This results in the following expectation for the execution times:

$$T_{backprop}(P, n, w, p) = \frac{p}{P} \cdot (w \cdot t_{weight} + n \cdot t_{neuron}) + t_{epo}(P, n, w)$$

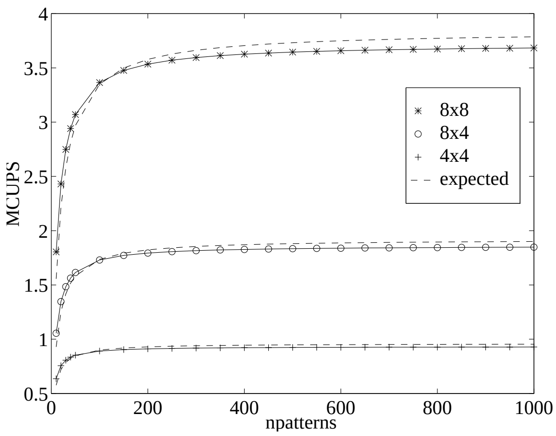
where t_{weight} and t_{neuron} for the three target platforms are listed below:

machine	t_{weight}	t_{neuron}
NSC	15.2	42.5
GCel	16.3	66.9
PX	1.01	3.9

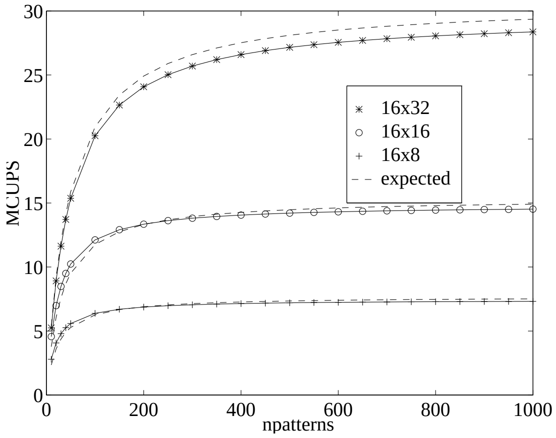
Table 6.5: *Fitted times for t_{weight} and t_{neuron} in μseconds.*

6.2.3 Performance of backpropagation

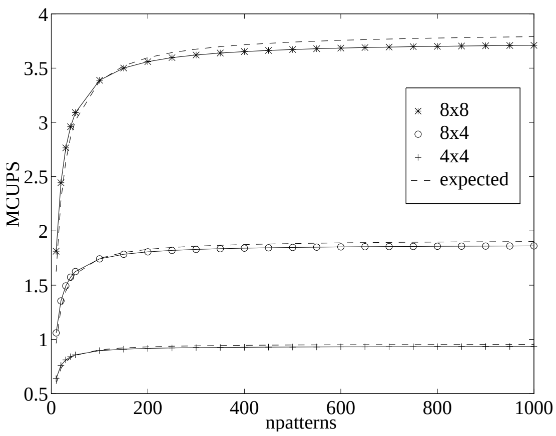
The parallel backpropagation simulations decomposed via dataset decomposition were measured as program kernels, i.e. the full application was timed. The measurements all involve the training times. Measurements were made for neural networks with a size in the range of 10 to 200K weights. For the large networks (>5K), the predictions are very good, in general within a deviation of 0% … 8%. As an example, consider Figures 6.2(a) to 6.2(d), which depict the measured and predicted performance for neural networks of size 100K and 200K weights, running on GCEL grids.



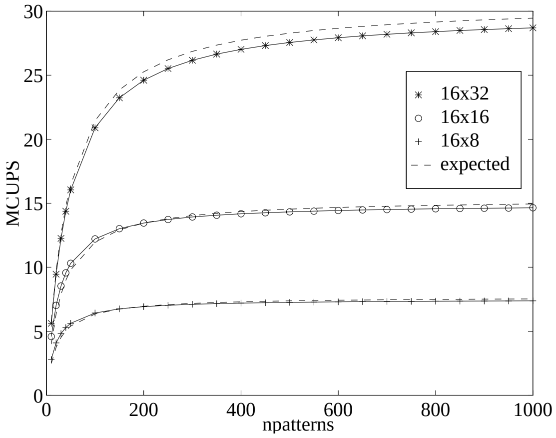
(a) *100K networks on small grids.*



(b) *100K networks on large grids.*



(c) *200K networks on small grids.*



(d) *200K networks on large grids.*

Figure 6.2: *Measured and expected performance for network decomposed backpropagation networks on GCel grids. The* `npatterns` *represent the number of patterns per processor.*

For all experiments using dataset decomposition, the plotted performance shows the characteristics as depicted in Figure 6.2. The larger the number of patterns that are computed during one epoch, the less influence the communication, gathering and update time represented by t_{epo} has on the performance. In the discussion held below, several tables are depicted containing the performance achieved for neural networks with diverse size and running on different processor systems. It will appear that not only the number of patterns, but also the size of the neural network determines the performance that is reached. This can also be explained in a straight forward manner. The larger the neural network, the more a processor is carrying out useful computations and the less influence is caused by the communication overhead. Both the number of patterns and the size of the neural network contribute to the problem size.

When considering the computation/communication ratio it is obvious that the more work each processor has to do, the more efficient the parallel program runs and thus the higher the performance that is reached. Using dataset decomposition, the performance for a backpropagation network is modeled via Equation (6.2). An upper bound for the maximal performance that can be reached on P processors is found by taking the limit for $p_{\to\infty}$ of Equation (6.2):

$$\mathcal{P}_{max}(P,n,w,p) = \lim_{p\to\infty} \frac{p\cdot w}{T_{pat}(P,n,w,p)+T_{epo}(P,n,w)}$$

Using the definitions of t_{pat} and t_{epo}, this can be rewritten to Equation (6.3), which for large w approximates to (6.4):

$$\begin{aligned}\mathcal{P}_{max}(P,n,w,p) &= \lim_{p\to\infty} \frac{p\cdot w}{p/P\cdot(w\cdot t_{weight}+n\cdot t_{neuron})+(w+nbias)\cdot t_{epo}} && (6.3)\\ &= \frac{P}{t_{weight}} && (6.4)\end{aligned}$$

The expected values of $\mathcal{P}_{max}$ for the different target platforms are given below:

machine	$\mathcal{P}_{max}(P)=\frac{P}{t_{weight}}$ **in** MCUPS							
	2x2	2x4	4x4	8x4	8x8	16x8	16x16	16x32
NSC	0.26	0.53	1.05	2.11	4.21	8.42	16.84	33.68
GCEL	0.25	0.49	0.98	1.96	3.93	7.85	15.71	31.41
PX	3.96	7.92	15.84	31.68	63.37	126.73	253.47	506.93

Table 6.6: *Maximum performance for backprop dataset decomposition.*

The next three subsections present the results of performance predictions achieved on respectively the GCel, PowerXPlorer and NSC.

6.2.4 Results for GCel

The results are presented in tables depicting the maximal performance in MCUPS and the minimum and maximum deviations in percent between measured and predicted performance. The achieved performances for the GCEL are depicted in Table 6.7. These were measured for networks of different size and a number of patterns varying from [1 ... 1000] patterns per processor. As mentioned before, all performances obtained for a certain neural network show the behavior as depicted in Figure 6.2. This means that the performance increases quickly in the domain of [1 ... 100] patterns per processor, after which it slowly increases towards its maximum for a larger number of patterns. The performance depicted in Table 6.7 almost reaches the predicted maximal performance shown in Table 6.6.

nweights	**gridsize**					
	4x4	8x4	8x8	16x8	16x16	16x32
10	0.2	0.4	0.8	1.7	3.3	6.6
50	0.4	0.8	1.7	3.3	6.6	13.2
100	0.5	1.0	2.0	4.0	8.0	15.9
500	0.7	1.4	2.7	5.5	10.9	21.6
1K	0.8	1.5	3.0	6.0	11.9	23.6
5K	0.9	1.7	3.4	6.8	13.4	26.6
10K	0.9	1.8	3.5	7.0	13.8	27.4
25K	0.9	1.8	3.6	7.1	14.2	28.0
50K	0.9	1.8	3.6	7.2	14.4	28.4
75K	0.9	1.8	3.7	7.3	14.5	28.6
100K	0.9	1.8	3.7	7.3	14.5	28.7
125K	0.9	1.9	3.7	7.3	14.6	28.8
150K	0.9	1.9	3.7	7.4	14.6	28.9
175K	0.9	1.9	3.7	7.4	14.6	28.9
200K	0.9	1.9	3.7	7.4	14.6	28.9

Table 6.7: *Performance in MCUPS for GCel.*

The deviations between measured and predicted results are depicted in Table 6.8. For the sequel of this thesis, all deviations between expected e_i and measured values m_i are given in percentages as the fraction $(e_i - m_i)/e_i \cdot 100.0\%$.

nweights	**gridsize**					
	4x4	8x4	8x8	16x8	16x16	16x32
10	7.7–16.9	3.3–16.9	0.1–16.8	2.3–16.9	0.9–16.5	1.4–16.1
50	0.1–2.4	0.4–2.4	0.2–3.6	0.6–6.9	0.0–8.5	0.1–10.9
100	1.4–1.5	1.4–3.3	1.4–4.3	1.5–6.3	1.2–7.2	1.3–8.9
500	0.3–1.7	1.1–1.8	1.8–1.8	1.8–3.3	1.9–3.9	1.9–5.1
1K	0.5–2.3	0.8–2.3	1.3–2.3	2.3–2.7	2.3–3.2	2.4–4.3
5K	0.7–2.8	0.4–2.8	0.8–2.8	2.0–2.8	2.4–2.9	2.9–3.3
10K	0.7–2.9	0.3–2.9	0.7–2.9	1.8–2.9	2.8–2.9	3.0–3.7
25K	1.9–5.2	2.4–5.1	2.9–5.4	3.4–5.4	3.8–5.6	4.2–5.6
50K	0.9–3.9	1.8–4.1	2.2–4.2	2.8–4.2	3.3–4.4	3.9–4.4
75K	0.4–3.2	1.2–3.3	1.7–3.3	2.5–3.4	2.9–3.6	3.6–3.6
100K	0.2–2.9	1.1–3.0	1.6–3.1	2.3–3.1	2.8–3.3	3.4–3.5
125K	0.0–2.6	0.9–2.7	1.4–2.7	2.2–2.9	2.7–3.0	3.0–3.4
150K	0.1–2.5	0.8–2.5	1.3–2.6	2.1–2.6	2.6–2.8	2.8–3.3
175K	0.2–2.4	0.7–2.5	1.3–2.5	2.1–2.6	2.6–2.7	2.7–3.3
200K	0.3–2.2	0.6–2.3	1.2–2.4	2.0–2.4	2.5–2.5	2.6–3.2

Table 6.8: *Min and max deviations for GCel (in %).*

6.2.5 Results for PX

The same set of experiments was carried out on the PowerXPlorer. The measured maximum performances and minimum and maximum deviations are depicted in Table 6.9.

nweights	**Performance**				**Deviations**			
	2x2	4x2	4x4	4x8	2x2	4x2	4x4	4x8
10	0.9	1.7	3.4	6.5	9.7–25.1	10.1–27.1	9.1–22.0	9.7–23.0
50	1.8	3.5	7.0	13.5	8.6–14.3	7.9–14.2	13.4–18.3	14.0–19.2
100	2.2	4.3	8.6	16.6	0.3–5.8	0.6–8.8	0.1–14.3	0.3–16.4
500	3.1	6.0	11.9	23.0	0.1–9.1	0.0–10.5	0.5–5.0	1.0–5.9
1K	3.4	6.7	13.1	25.3	0.1–11.0	0.0–13.4	1.0–4.3	1.2–5.0
5K	3.8	7.4	14.4	27.9	0.5–10.2	0.8–15.0	3.5–7.1	3.7–7.2
10K	3.9	7.6	14.8	28.7	0.0–9.8	1.5–14.8	3.6–8.1	3.8–7.9
25K	3.9	7.7	15.2	29.5	0.2–1.7	0.1–1.2	0.2–1.0	0.0–1.7
50K	3.9	7.9	15.4	30.0	0.9–3.1	0.3–2.6	0.1–2.4	0.2–2.2
75K	3.9	7.9	15.6	30.2	1.3–3.9	0.6–3.4	0.2–3.2	0.0–3.1
100K	3.9	7.9	15.6	30.3	1.4–4.4	0.7–3.9	0.2–3.7	0.2–3.5
125K	3.9	7.9	15.7	30.4	1.6–4.7	0.9–4.2	0.1–4.0	0.1–3.8
150K	3.9	7.9	15.7	30.5	1.6–4.9	0.9–4.3	0.0–4.2	0.0–4.0
175K	3.9	7.9	15.7	30.5	1.7–4.9	0.9–4.4	0.0–4.3	0.0–4.1
200K	3.9	7.9	15.7	30.5	1.8–5.0	1.0–4.5	0.1–4.4	0.1–4.1

Table 6.9: *Performance (MCUPS) and deviation range (%) for PX.*

6.2.6 Results for NSC grids and trees

The backpropagation neural networks were also measured on NSC grids and trees. The same programs were used as the ones that were utilized for GCel and PX grids. However for tree topologies another communication layer implementing the gathering and broadcasting of data was exploited, as explained in Chapter 5.

nweights	**Performance**			**Deviations**		
	4x4	8x4	8x8	4x4	8x4	8x8
10	0.3	0.5	1.0	9.9–11.1	14.3–14.6	13.8–14.1
50	0.5	1.0	2.0	2.6–19.8	7.9–25.5	7.9–26.4
100	0.7	1.2	2.4	1.4–13.4	0.0–21.0	0.5–22.8
500	0.9	1.6	3.2	3.8–6.2	8.6–12.3	8.7–12.4
1K	0.9	1.7	3.5	0.1–2.3	2.8–6.7	2.7–7.0
5K	1.0	1.9	3.8	3.9–4.0	1.3–2.4	1.3–1.8
10K	1.0	2.0	3.9	3.8–4.6	1.4–1.7	0.9–1.3
25K	1.0	1.9	3.9	3.2–7.1	1.2–4.3	0.0–0.9
50K	0.9	1.8	4.0	2.0–9.6	2.3–6.9	0.5–3.3
75K	0.9	1.9	3.7	0.8–4.5	0.7–4.6	3.6–5.7
100K	0.9	1.9	3.9	0.9–4.6	0.2–1.9	0.0–0.8
125K	0.9	1.8	3.6	0.4–6.4	4.4–9.1	4.2–8.0
150K	0.9	1.7	3.6	0.4–6.5	5.5–10.9	3.6–7.2
175K	0.9	1.8	3.8	0.2–3.3	3.4–7.4	1.2–2.4
200K	0.9	1.9	3.9	0.3–3.2	0.5–4.2	0.2–0.5

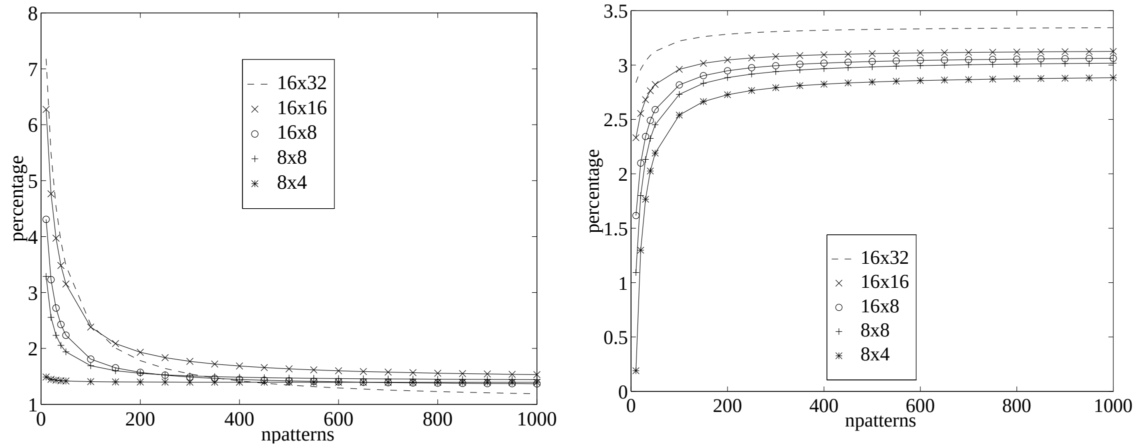
Table 6.10: *Performance (MCUPS) and deviation range (%) for NSC grids.*

nweights	Performance				Deviations			
	4	13	40	63	4	13	40	63
10	0.1	0.2	0.6	1.0	0.6–16.1	0.0–10.3	0.1–7.5	0.1–9.5
50	0.1	0.4	1.3	2.1	5.2–11.8	2.6–7.2	2.1–5.4	4.1–7.4
100	0.1	0.5	1.5	2.4	0.2–4.7	0.0–4.2	0.0–7.1	0.0–5.7
500	0.2	0.6	2.0	3.2	2.9–8.0	1.9–8.6	0.8–8.4	1.0–8.3
1K	0.2	0.7	2.1	3.4	0.9–2.0	0.2–2.7	0.1–3.2	0.0–4.2
5K	0.2	0.7	2.3	3.8	0.6–2.1	1.2–2.9	1.2–3.1	1.3–1.9
10K	0.2	0.7	2.4	3.9	0.6–2.5	1.3–3.7	1.3–4.2	1.4–3.0
25K	0.2	0.8	2.5	4.0	2.1–4.2	2.7–4.9	2.8–6.5	2.9–6.9
50K	0.2	0.8	2.5	4.0	1.1–3.5	1.8–4.3	1.8–6.0	1.9–6.4
75K	0.2	0.8	2.5	4.0	0.5–3.0	1.1–4.0	1.2–5.6	1.4–6.1
100K	0.2	0.8	2.5	4.0	0.3–2.8	1.0–3.6	1.0–5.5	1.1–6.0
125K	0.2	0.8	2.5	4.0	0.1–2.5	0.7–3.6	0.8–5.3	0.9–5.7
150K	0.2	0.8	2.5	4.0	0.0–2.4	0.5–3.4	0.6–5.3	0.8–5.4
175K	0.2	0.8	2.5	4.0	0.0–2.2	0.5–3.2	0.6–5.0	0.6–4.2
200K	0.2	0.8	2.5	4.0	0.0–2.3	0.4–3.1	0.5–4.9	0.5–3.8

Table 6.11: *Performance (MCUPS) and deviation range (%) for NSC trees.*

6.3 Discussion of the results

The tables depicted above present the achieved performances and deviations between the measured and predicted performance. For all results it appears that the predicted maximal performance is reached for large problem sizes. It can also be observed that in general the larger the problem size, the better the predictions. Note that not only the number of weights, but also the number of patterns contributes to the problem size. The typical behavior of the deviations for a given neural network related to the number of patterns is as depicted below in Figure 6.3.



(a) *Deviations for 100 weights.* (b) *Deviations for 100K weights.*

Figure 6.3: *Typical deviations on different GCel grids (in %).*

This behavior can be explained as follows. It was noted that – similar like the communication time for small messages – the calculation times depend heavily on the size of the neural network. Using the performance prediction model for dataset decomposition, two parameters are estimated, the times t_{pat} and t_{epo}. For neural networks > 10K weights, these times are estimated using Table 6.5. For smaller networks, a fit of the measured function kernels was required in order to arrive at proper performance predictions. Consider the performance model, given an expected execution time for a certain neural network on a certain processor topology $t_{exp} = p \cdot t_{pat} + t_{epo}$ and assuming that the measured time can be modeled via two quality factors (f_1, f_2) such that $t_{meas} = p \cdot t_{pat} \cdot f1 + t_{epo} \cdot f2$. The deviation (fraction) is given as:

$$\begin{aligned} dev(f_1, f_2) &= \frac{t_{exp} - t_{meas}}{t_{exp}} \\ &= \frac{(1 - f_1) \cdot p \cdot t_{pat} + (1 - f_2) \cdot t_{epo}}{p \cdot t_{pat} + t_{epo}} \end{aligned}$$

The derivative of $dev(f_1, f_2)$ equals:

$$dev'(f_1, f_2) = -\frac{t_{pat} \cdot t_{epo} \cdot (f_1 - f_2)}{(p \cdot t_{pat} + t_{epo})^2}$$

If f_1 equals f_2, the deviation has a constant value (its derivative is zero). Note that p, t_{pat}, t_{epo} are all greater than zero. As the sign of the derivative determines the slope of the deviation, by considering the plotted deviations as in Figure 6.3 it can be determined that $f_1 > f_2$ or vice versa. Evaluating all deviations of the results, it appears that for networks < 5K weights, the expected values for t_{comm} have a larger deviation than the expected values for t_{pat}. This can be derived from the fact that the derivative is negative, so $f1 > f2$[1]. In other words, it can be concluded that for small neural networks communication not only forms a bottle neck for the performance, but also that it is the main source for the deviations of the performance prediction model. For large neural networks, the expected maximum performance is reached and the predictions are within a number of percentages. As for these networks the communication becomes less important, possible differences in the calculation times rule the shape of the deviations.

6.4 Dataset decomposition for Kohonen

The sequential KSOM (Kohonen Self-Organizing feature Map) algorithm and the identification of its function kernels is described in Chapter 4, Section 4.2.2.1. Similar to the dataset decomposed version of the backpropagation algorithm, the version of the KSOM algorithm differs from its sequential equivalent in the way in which the weights are updated. During one epoch, for each of the patterns stored locally on a processor, the weight changes are computed. Subsequently, they are gathered, accumulated and broadcast after

[1]This holds for $f1, f2 \leq 1$. For other cases a similar consideration can be made.

which the weight updating takes place. The dataset decomposed version of the KSOM algorithm is depicted in Algorithm 6.4:

```
for (epoch=0;error>errcrit&&epoch<nepochs;epoch++) {
   for (p=0;p<npatterns;p++) {
     c = FindWinner(p);                  /* compute_net_status() */
     for (i=0;i<nneurons;i++)            /* compute_net_changes()*/
        if (Neighbour(c,i))              /*         ...          */
          ComputeWeightChanges(p,c,i);   /*         ...          */
   }
   GAB_WeightChanges();
   UpdateWeightChanges();
}
```

Algorithm 6.4: *Algorithm for KSOM dataset decomposition.*

From this algorithm, the function kernels can be identified as in Table 6.12.

kernel	**time required**
`FindWinner()`	t_{fnd}
`Neighbour()`	t_{nei}
`ComputeWeightChanges()`	t_{dw}
`GABWeightsChanges()`	$T^{\mathtt{GAB}}(P, nweights)$
`UpdateWeightChanges()`	t_{chw}

Table 6.12: *Function and communication kernels for Kohonen decomposed via dataset decomposition.*

In Section 4.2.2.1 it is discussed that for determining the total execution time, the times for the routines `FindWinner()` and `UpdateWeightChanges()` must be multiplied by the number of weights. The same holds for `GABWeightsChanges()` and `UpdateWeightChanges()`, but the execution time for `Neighbour()` is related to the number of neurons in the Kohonen map. Given a winning neuron, for all neurons that lay within its neighborhood, the weight changes have to be computed. In our implementation of the KSOM neural network, it is decided whether two neurons are neighbors by checking if their Euclidean distance is within a certain range r. The upper bound on the number of neurons that have to compute their weight changes is given by $\pi \cdot r^2$. Initially, $r = \sqrt{n}/4$, where n is the number of neurons. The total calculation time per pattern can thus be bounded by t_{pat}, which is given in Equation (6.5). Note that the number of weights $w = N \cdot n$, where N is the dimension of the input space. The extra time required per epoch is given in (6.6):

$$\begin{aligned} t_{pat} &= n \cdot t_{nei} + w \cdot t_{fnd} + N \cdot \pi \cdot (\sqrt{n}/4)^2 \cdot t_{dw} \\ &= n \cdot t_{nei} + w \cdot t_{fnd} + N \cdot \pi \cdot n/16 \cdot t_{dw} \\ &= n \cdot t_{nei} + w \cdot (t_{fnd} + \pi/16 \cdot t_{dw}) \qquad (6.5) \\ t_{epo} &= w \cdot t_{chw} + T^{\mathtt{GAB}}(P, w) \qquad (6.6) \end{aligned}$$

6.4.1 Measurements for function kernels

The function kernels listed in Table 6.12 were measured on the three target platforms. The different values for t_{nei}, t_{fnd}, etcetera were determined as described in Chapter 4. The results were measured for different network settings (N, n), where each result represents the outcome of a sequential program kernel measurement, i.e. for a certain network, the Algorithm 6.4 was measured without executing `GABWeightsChanges()`. Instead of the backpropagation network, it is possible to identify only one code fragment kernel (the computation of the neighbors). Therefore, Equation (6.5) can also be used for small neural networks and no fitting is required. The measured times for the function kernels are depicted in 6.13.

kernel	**NSC**	**GCEL**	**PX**
tnei	24.95	20.77	3.37
tfnd	8.13	6.77	.474
tdw	10.01	8.34	.520
tcw	6.40	5.33	.335

Table 6.13: *Times for KSOM function kernels (μseconds).*

Using these results and Equations (6.5) and (6.6), the maximum performance for KSOM networks can be estimated as Equation (6.7), for which the computed values are given in Table 6.14.

$$\begin{aligned}\mathcal{P}_{max}(P, n, w, p) &= \lim_{p\to\infty} \frac{p \cdot n \cdot N}{p \cdot n/P \cdot (t_{nei} + N \cdot (t_{fnd} + \pi/16 \cdot t_{dw}) + T_{epo}(P, n, w)} \\ &= P \cdot N/(t_{nei} + N \cdot (t_{fnd} + \pi/16 \cdot t_{dw})) \qquad (6.7)\end{aligned}$$

machine	$\mathcal{P}_{max}$ **in** MCUPS							
	2x2	2x4	4x4	8x4	8x8	16x8	16x16	16x32
NSC	0.38	0.77	1.54	3.07	6.15	12.30	24.60	49.20
GCEL	0.46	0.92	1.85	3.69	7.38	14.77	29.54	59.07
PX	6.47	12.94	25.88	51.76	103.52	207.04	414.09	828.18

Table 6.14: *Maximum performance for KSOM dataset decomposition.*

6.4.2 Results for KSOM

Below, the measured and maximum performance and deviations for the KSOM networks decomposed via dataset decomposition are given. The size of the networks is varied from 10 to 200K weights. The number of patterns was limited to 1000 per processor. Full program kernels were measured, similar to the backpropagation networks.

nweights	gridsize					
	4x4	8x4	8x8	16x8	16x16	16x32
10	0.6	1.2	2.5	4.9	9.8	19.3
50	0.7	1.3	2.7	5.3	10.6	21.1
100	1.0	2.1	4.2	8.3	16.4	32.4
500	1.1	2.3	4.5	9.0	17.8	35.2
1K	1.4	2.8	5.7	11.2	22.3	43.9
5K	1.4	2.8	5.5	11.0	21.8	42.9
10K	1.6	3.2	6.3	12.6	24.9	48.9
25K	1.5	3.1	6.6	12.1	22.3	45.4
50K	1.6	3.1	6.3	12.4	24.6	48.3
75K	1.7	3.3	6.6	13.0	25.8	50.6
100K	1.7	3.4	6.7	13.4	26.4	51.9
125K	1.7	3.4	6.9	13.6	26.8	52.7
150K	1.8	3.5	6.9	13.7	27.1	53.2
175K	1.8	3.5	7.0	13.8	27.3	53.6
200K	1.8	3.5	7.0	13.9	27.5	53.9

Table 6.15: *Performance (MCUPS) for GCel.*

nweights	gridsize					
	4x4	8x4	8x8	16x8	16x16	16x32
10	5.9–19.3	1.6–19.2	1.1–19.0	1.1–18.8	1.6–18.5	0.4–18.1
50	1.0–1.2	1.0–1.6	1.0–1.7	1.0–2.1	1.0–2.1	0.7–1.0
100	1.7–2.4	2.0–2.4	1.8–2.4	2.0–2.4	1.9–2.4	0.1–2.3
500	1.6–2.9	1.8–2.9	1.5–2.9	1.7–2.9	1.4–2.9	0.1–2.8
1K	2.1–4.2	2.2–4.2	1.8–4.2	1.9–4.2	1.6–4.1	0.2–4.0
5K	2.0–4.1	2.1–4.1	1.7–4.1	1.8–4.1	1.5–4.1	0.2–3.9
10K	2.3–4.9	2.3–4.9	1.9–4.9	1.9–4.8	1.6–4.8	0.2–4.6
25K	0.1–0.5	0.0–0.1	2.9–8.3	0.0–0.5	2.2–7.0	0.0–3.5
50K	3.8–7.5	3.5–7.5	2.8–7.5	2.6–7.5	2.1–7.4	0.1–7.2
75K	3.1–6.4	2.9–6.4	2.3–6.3	2.2–6.3	1.8–6.3	0.3–6.1
100K	2.7–5.8	2.6–5.8	2.1–5.8	2.0–5.8	1.7–5.7	0.3–5.5
125K	2.5–5.5	2.4–5.4	1.9–5.4	1.9–5.4	1.6–5.3	0.2–5.1
150K	2.3–5.2	2.3–5.2	1.8–5.2	1.8–5.1	1.5–5.1	0.2–4.9
175K	2.2–5.0	2.2–5.0	1.7–5.0	1.8–4.9	1.5–4.9	0.1–4.7
200K	2.1–4.9	2.2–4.8	1.7–4.9	1.7–4.8	1.5–4.8	0.1–4.6

Table 6.16: *Min and max deviations (%) for GCel.*

nweights	performance				deviations			
	2x2	4x2	4x4	4x8	2x2	4x4	4x4	4x8
10	1.5	2.9	5.7	10.8	0.7–12.8	1.3–15.9	1.7–16.2	2.5–16.8
50	1.6	3.2	6.2	12.3	3.0–4.7	2.1–4.6	1.1–4.8	0.3–4.5
100	2.8	5.6	11.1	21.5	1.9–4.4	0.4–4.3	0.2–4.0	0.1–4.0
500	2.9	5.8	11.4	22.3	2.0–3.9	0.3–3.7	0.2–3.7	0.1–3.6
1K	4.2	8.2	16.1	31.2	1.8–3.6	0.1–3.6	0.2–3.7	0.1–3.5
5K	4.3	8.4	16.5	32.0	2.0–3.8	0.4–3.6	0.1–3.7	0.0–3.5
10K	5.3	10.3	20.2	39.0	1.7–3.4	0.1–3.7	0.1–3.6	0.1–3.3
25K	4.1	8.1	15.8	30.6	2.5–5.7	0.4–5.0	0.0–6.3	0.1–6.1
50K	5.0	9.9	19.7	38.4	2.3–5.9	0.2–5.1	0.0–3.9	0.4–2.3
75K	5.6	10.9	20.9	41.2	1.2–2.7	0.2–3.1	0.3–5.3	0.4–2.4
100K	5.8	11.4	22.0	42.9	1.6–4.1	0.2–3.1	0.1–4.2	0.2–2.2
125K	5.8	11.5	22.8	44.0	2.0–5.5	0.1–4.1	0.1–2.8	0.1–2.0
150K	6.1	12.0	22.7	44.5	1.0–2.4	0.1–1.9	0.0–4.8	0.3–2.5
175K	6.2	11.8	23.4	44.2	1.0–2.3	0.0–4.7	0.0–3.3	0.1–4.3
200K	6.2	12.0	23.4	45.6	1.4–3.6	0.2–4.0	0.2–4.1	0.1–2.2

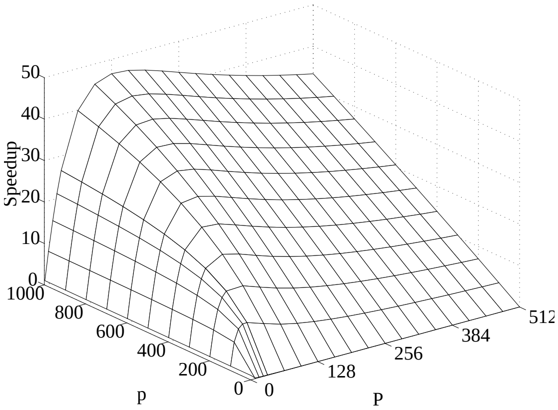
Table 6.17: *Performance (MCUPS) and deviation range (%) for PX.*

nweights	performance			gridsize		
	4x4	8x4	8x8	4x4	8x4	8x8
10	0.6	1.1	2.1	0.2–9.0	0.0–9.3	0.0–7.9
50	0.6	1.3	2.5	2.7–5.8	2.9–6.1	2.9–5.7
100	1.0	2.0	4.0	0.3–5.2	0.4–5.9	0.4–5.7
500	1.1	2.2	4.5	0.0–4.5	0.0–4.8	0.0–3.6
1K	1.4	2.9	5.7	0.1–3.0	0.3–3.8	0.1–1.9
5K	1.4	2.8	5.6	0.0–2.0	0.1–2.5	0.0–1.4
10K	1.6	3.2	6.4	0.4–3.1	0.4–3.1	0.3–3.1
25K	1.2	2.5	4.8	3.9–6.5	0.1–4.0	2.7–5.3
50K	1.4	2.7	5.3	2.8–5.0	0.9–5.7	3.1–6.1
75K	1.4	2.8	5.6	2.0–3.8	0.5–4.8	2.6–5.5
100K	1.4	2.9	5.8	3.0–5.5	0.8–4.3	1.4–2.7
125K	1.5	3.0	5.8	1.9–3.7	0.2–2.5	2.1–3.9
150K	1.5	3.0	5.8	1.0–2.3	0.0–3.3	2.5–5.3
175K	1.5	3.0	5.8	2.6–4.9	0.6–3.8	2.6–5.5
200K	1.5	3.0	6.0	2.1–4.1	0.7–4.6	1.7–2.8

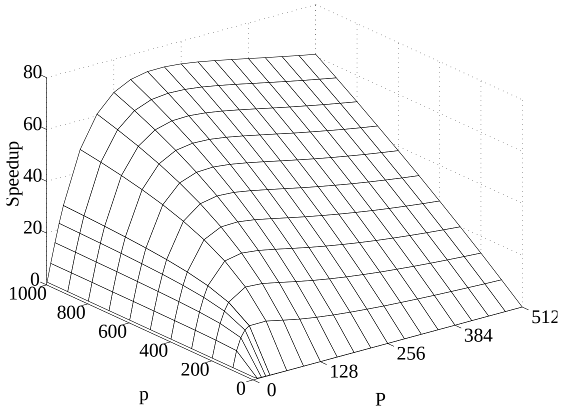
Table 6.18: *Performance (MCUPS) and deviation range (%) for NSC.*

6.5 Fixed-size speedup

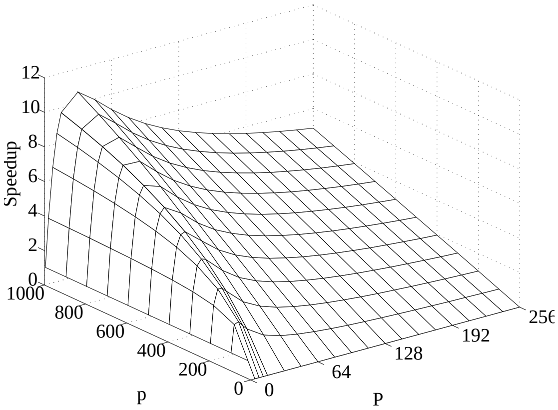
The previous sections show that for dataset decomposition, the performance prediction model is able to predict the execution times for a given neural network architecture and processor topology. Especially for larger problem sizes, the accuracy of the predictions is good. Using the predicted execution times, the speedup, efficiency and scalability can be predicted within the same accuracy. As already can be observed in the tables listed above, for large problems linear speedups can be reached. Consider for example any of the tables that depict the maximum performance. The bottom rows of each table represent the larger neural networks. Following the values of a row from left to right, the performance increases each step with a factor of two. As the number of processors also increases with a factor of two, the speedup is linear. As also can be observed, the speedups for smaller problems (small number of patterns p) are not linear. This is because smaller problems suffer from communication overheads. The speedups mentioned here represent the scaled speedup, as the problem size for a given network is ruled by the number of patterns *per processor*. Though, as expected, the achieved scalabilities are high, the question that is discussed in this section is to find out at which point the (fixed-size) speedup drops, i.e. for which problem size, for which number of processors, does the speedup no longer increase but rather decrease. This point is known as the speedup limit. In the next section, this issue is discussed for scaled speedups, i.e. the scalability limit is searched. Plotting the fixed-size speedup for different neural networks sizes and number of patterns gives similar results for all three configurations (NSC, GCEL and PX). For example consider the speedups depicted below. For backpropagation networks of size 10 and 200K weights, for a varying number of patterns, the expected speedup is depicted for grid and tree topologies.



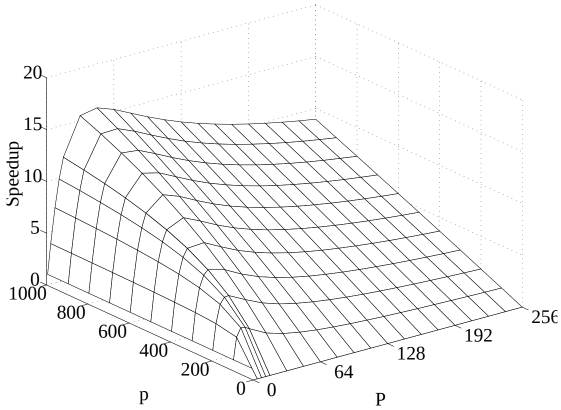
(a) *GCEL grids, w = 10.*



(b) *GCEL grids, w= 200K*



(c) *PX grids, w = 10.*



(d) *PX grids, w= 200K*

Figure 6.4: *Speedups for GCEL and PX grid topologies.*

These plots are computed using the definition of fixed-size speedup:

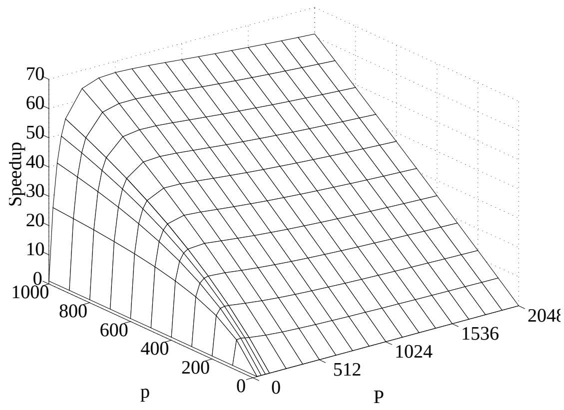
$$S(P, n, w, p) \;\; = \;\; \frac{T(1, n, w, p)}{T(P, n, w, p)}$$

which is defined for dataset decomposition as:

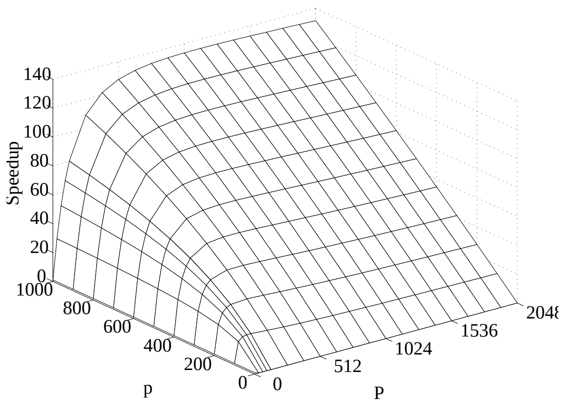
$$S(P, n, w, p) \;\; = \;\; \frac{p \cdot t_{pat}(n, w) + t_{chg}(n, w)}{p/P \cdot t_{pat}(n, w) + t_{chg}(n, w) + T^{GAB}(P, n, w)}$$

where the times for communication for respectively a grid and tree topology are:

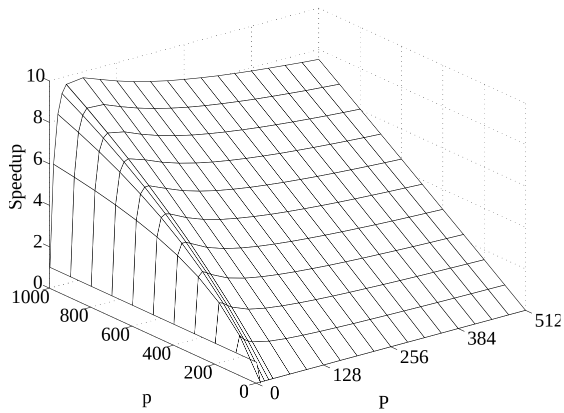
$$T^{GAB}_{grid}(P, n, w) = 2 \cdot (\sqrt{P} - 1) \cdot (2 \cdot t_{comm}(n, w) + t_{acc}(n, w))$$
$$T^{GAB}_{tree}(P, n, w) = 3 \cdot log^3(P) \cdot (2 \cdot t_{comm}(n, w) + t_{acc}(n, w))$$



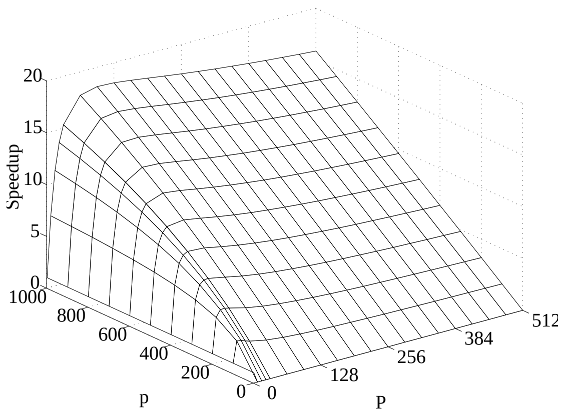
(a) *GCEL trees, w = 10.*



(b) *GCEL trees, w= 200K*



(c) *PX trees, w = 10.*



(d) *PX trees, w= 200K*

Figure 6.5: *Speedups for GCEL and PX tree topologies.*

As can be observed in these figures, for both small and large neural networks, and for all patterns in the range [1 ... 1000] per processor, there is a point at which the speedup reaches the speedup limit. Furthermore it is noted that for most cases the speedup as well as the speedup limit for a tree are higher than those for a grid. Only for larger networks on the PX, the grid seems to be in favor of a tree topology (compare Figures 6.4(c) and 6.5(c)). On the other hand, in these figures it shows that for larger processor systems, the speedup is higher for a tree.

6.5.1 The speedup limit

If the speedup limit is reached for a certain number of processors, this number can be found at the point where the derivative for the speedup equals zero (see also [114]). Taking the derivative $\frac{\partial S}{\partial P}$ and computing P for $\frac{\partial S}{\partial P} = 0$ gives for a grid:

$$\begin{aligned}
\frac{\partial S}{\partial P} &= 0 \;\Rightarrow\; \frac{2 \cdot t_{comm}(n,w) + t_{acc}(n,w)}{\sqrt{P}} - \frac{p \cdot t_{pat}(n,w)}{P^2} = 0 \\
&\Rightarrow\; P^{grid}_{max} = \left(\frac{p \cdot t_{pat}(n,w)}{2 \cdot t_{comm}(n,w) + t_{acc}(n,w)} \right)^{\frac{2}{3}} \qquad (6.8)
\end{aligned}$$

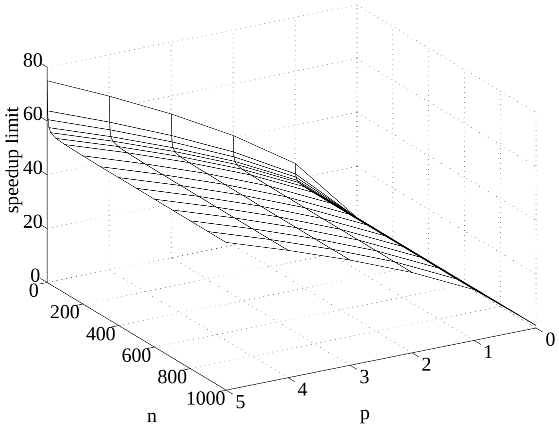
and for a tree, where we estimate the depth of a tree with $log^3(P)$:

$$\begin{aligned}
\frac{\partial S}{\partial P} &= 0 \;\Rightarrow\; \frac{3 \cdot (2 \cdot t_{comm}(n,w) + t_{acc}(n,w))}{P \cdot ln(3)} - \frac{t_{pat}(n,w)}{P^2} = 0 \\
&\Rightarrow\; P^{tree}_{max} = \frac{ln(3) \cdot p \cdot t_{pat}(n,w)}{3 \cdot (2 \cdot t_{comm}(n+w) + t_{acc}(n+w))} \qquad (6.9)
\end{aligned}$$

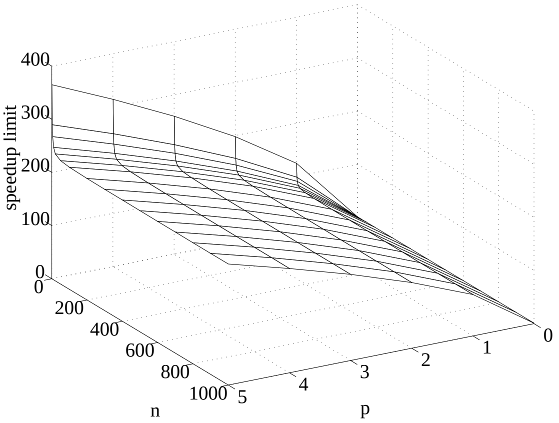
Intuitively, such a relation between the number of patterns, the calculation and the communication times could be expected. Indeed, if the communication times are high compared to the calculation times, the simulations do not run efficiently and the speedup limit is reached relatively soon. Similarly, if the number of patterns increases, the calculation time becomes more important and the speedup limit raises. For a small number of patterns, the network size is of importance for the speedup limit. For example, if every processor computes only one pattern, communication is required after the computation of each pattern. For small networks, especially for a fast machine like the PX, the calculation times are relatively low compared to the communication times. These principles can be observed in the discussions about Kohonen and backpropagation networks given below.

6.5.2 Fixed-size speedups for backpropagation

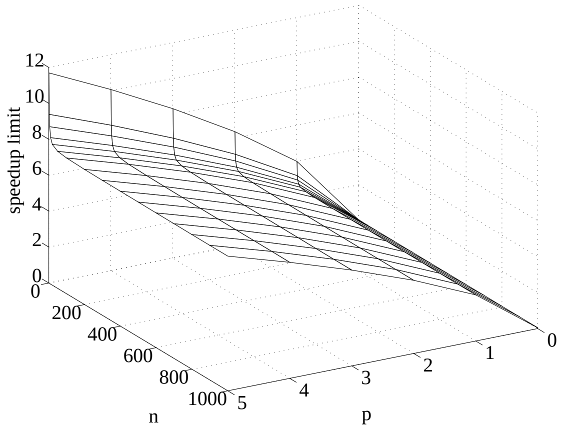
Using Equations (6.8) and (6.9), for a given neural network and a given number of patterns, the speedup limit can be computed. In order to determine the effect of network size and number of patterns on the speedup limit, consider the figures below. Plotted is the maximum number of processors after which the speedup drops, computed via these equations.



(a) *Speedup limit for NSC.*



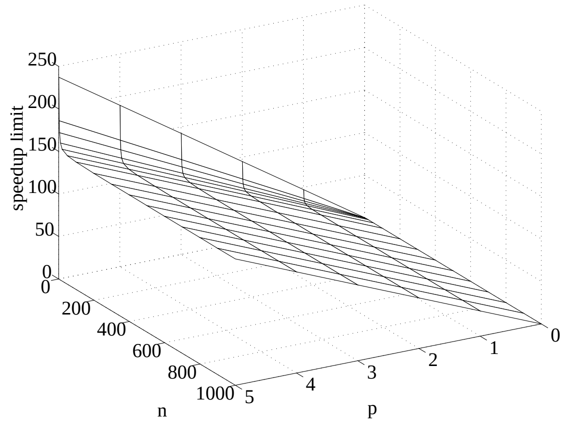
(b) *Speedup limit for GCEL.*



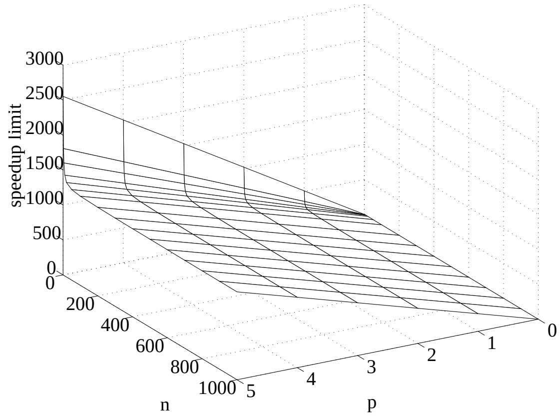
(c) *Speedup limit for PX.*

Figure 6.6: *Speedup limit for backpropagation on transputer grids. The y-axis gives the network size in terms of number of neurons, the x-axis represents the number of patterns per processor. This means that for the GCEL,* $p = 5$ *means that the number of patterns in the processor network is* $5 \cdot 512$*, and similarly for the NSC and PX, that this number is* $5 \cdot 64$ *and* $5 \cdot 32$*.*

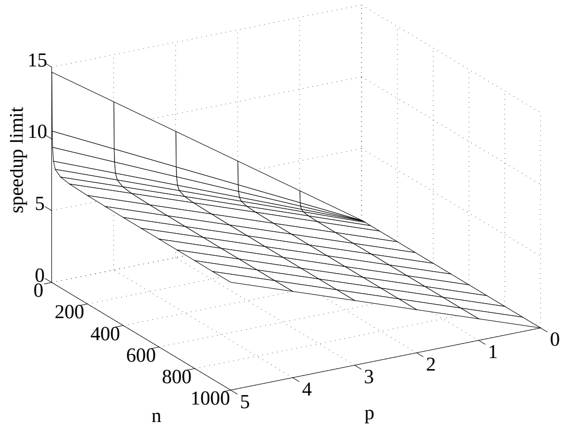
Considering these plots, it shows that the most important parameter is the number of patterns. The larger this number, the higher the speedup limit. To compare this to the speedup limit for tree topologies, consider Figure 6.7. As there is a linear relation between the number of patterns p and the times for calculation and communication, compared to the $power(2/3)$ for grids, it is obvious that trees provide a more efficient target platform for dataset decomposed neural networks than grids.



(a) *Speedup limit for NSC.*



(b) *Speedup limit for GCEL.*



(c) *Speedup limit for PX.*

Figure 6.7: *Speedup limit for backpropagation on transputer trees.*

When considering these plots, it is noted that for larger neural networks, the speedup limit approximates some asymptote:

$$\lim_{w\to\infty} P_{max} = \left(\frac{p \cdot t_{weight}}{2 \cdot t_{comm} + t_{acc}}\right)^{\frac{2}{3}} \quad \text{for grids} \tag{6.10}$$

$$\text{or} \quad \frac{ln(3) \cdot p \cdot t_{weight}}{3 \cdot (2 \cdot t_{comm} + t_{acc})} \quad \text{for trees} \tag{6.11}$$

For respectively the NSC, GCEL and PX systems, the speedup limit for 5 patterns per processor computed for these equations becomes 55, 227 and 7, (and 148, 1256, 8 for trees), which corresponds with the figures depicted above. Considering that these numbers are met relatively soon, this means that even for neural networks with moderate size ($n = 100$), the speedup limit can be computed in this way. Furthermore, this means that for a given number of patterns, the speedup limit decreases to (6.10) or (6.11) when increasing the network size.

6.5.3 Fixed-size speedups for Kohonen networks

Equations (6.8) and (6.8) also hold for Kohonen neural networks. Substituting t_{pat}, etcetera for Kohonen, and dividing numerator and denominator by the number of neurons n (see Equations (6.5) and (6.6)), the speedup limit becomes:

$$\begin{aligned} P^{grid}_{max} &= \left(\frac{p \cdot (t_{nei} + N \cdot (t_{fnd} + \frac{\pi}{16} \cdot t_{dw}))}{2 \cdot t_{comm}(N) + t_{acc(N)})} \right)^{\frac{2}{3}} \\ P^{tree}_{max} &= \frac{ln(3) \cdot p \cdot (t_{nei} + N \cdot (t_{fnd} + \frac{\pi}{16} \cdot t_{dw})}{3 \cdot (2 \cdot t_{comm}(N) + t_{acc}(N))} \end{aligned}$$

This means that the speedup limit for Kohonen networks depends on the number of inputs N and the number patterns. Similar plots can be made as depicted above for backpropagation. When increasing the network size for small networks (small N), the drop in speedup limit that was observed for backpropagation is even more dramatic. Again, by taking the limit of the network size, P_{max} can be found as:

$$\begin{aligned} \lim_{N \to \infty} P^{grid}_{max} &= \left(\frac{p \cdot (t_{fnd} + \frac{\pi}{16} \cdot t_{dw})}{2 \cdot t_{comm} + t_{acc}} \right)^{\frac{2}{3}} \\ \lim_{N \to \infty} P^{tree}_{max} &= \frac{p \cdot ln(3) \cdot (t_{fnd} + \frac{\pi}{16} \cdot t_{dw})}{3 \cdot (2 \cdot t_{comm} + t_{acc})} \end{aligned} \tag{6.12}$$

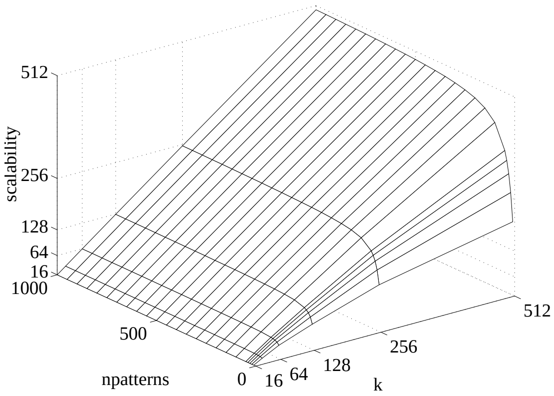
For respectively the NSC, GCEL and PX systems, the speedup limit for 5 patterns per processor becomes 35, 146 and 6 (and 74, 648, 6 for trees). Note that these numbers are somewhat lower than the speedup limits found for backpropagation in the previous section. This is caused by the fact that for Kohonen, for the same number of weights, the calculation times are less than for backpropagation.

6.6 Scalability and efficiency

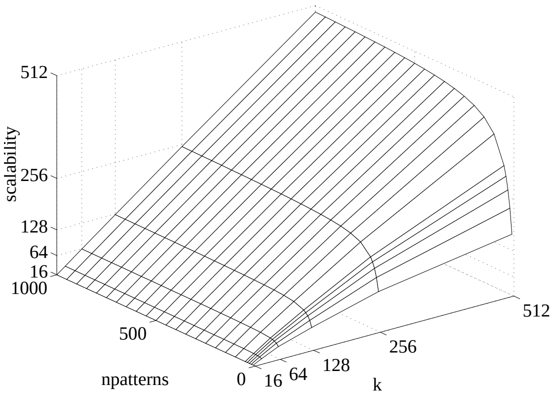
The scalability issue is important if someone scales up his problem with a certain factor k and hopes to solve it in the same time as the original problem was solved on P processors by increasing P to $k \cdot P$. The scalability is k, if both problems indeed can be solved in the same time, or more formally:

$$\begin{aligned} S^{scal}(k, P, (n, w, p)) &= k \cdot f^{scal}(k, P, (n, w, p)) \quad \text{where} \\ f^{scal}(k, P, (n, w, p)) &= \frac{T(P, n, w, p)}{T(k \cdot P, k \cdot (n, w, p))} \end{aligned} \tag{6.13}$$

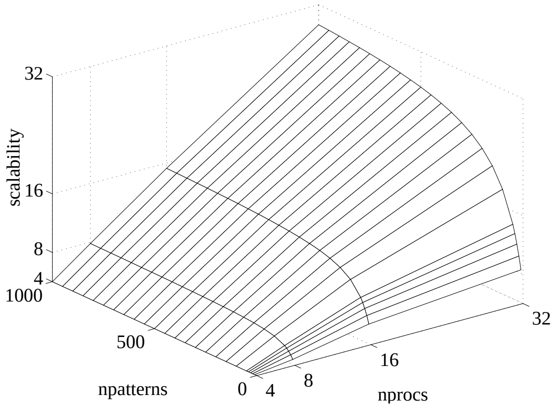
Scaling up the problem via $k \cdot (n, w, p)$ can be done by either increasing the network size or by using more patterns. For example, consider Figure 6.8. The plots represent the measured scalability for small and large neural networks, for a varying number of patterns. The execution time for a problem with 10 or 200K weights computing p patterns was measured on one processor. Furthermore, the same networks computing $k \cdot p$ patterns were measured on k processors. Plotted are the divided execution times multiplied by k; which equals the scalability.



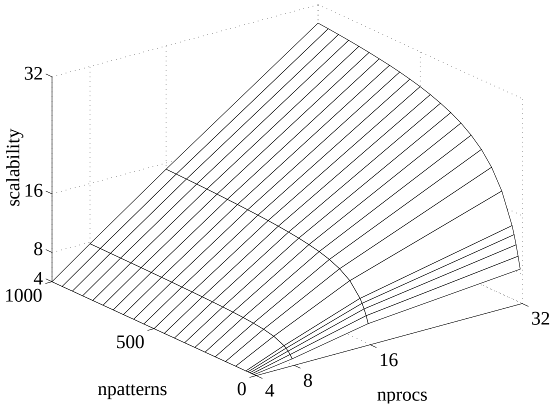
(a) *Scalability for 10 weights (GCEL).*



(b) *Scalability for 200K weights (GCEL).*



(c) *Scalability for 10 weights (PX).*



(d) *Scalability for 200K weights (PX).*

Figure 6.8: *Scalability for backpropagation networks on GCEL and PX grids.*

As can be observed, the scalability limit is not reached. Not for small networks of 10 weights and not for large networks of 200K weights. Similar as described in the previous section, this limit can be found by finding the number of processors for which $\frac{\partial S^{scal}}{\partial P}$ equals zero. The expectation is that, – if such a limit exists –, it must be much higher than the fixed-size speedup limit, as the problem size is scaled linearly with the number of processors. Computing $\frac{\partial S^{scal}}{\partial P}$ gives no reasonable P for which the derivative equals zero. Considering Figure 6.8 and plotting the scalability for varying k, n, p, and P, it can be concluded that for dataset decomposition no scalability limit exists. However, for large k, the scalability factor approaches zero, in particular if p is small compared to k. This means that a large number of processors are added without any significant gain in execution time; a large amount of resources is wasted. The part of the processors that is being used efficiently is defined by the *efficiency*. The definition of efficiency is speedup divided by the number of processors. This means that the higher the speedup, the higher the efficiency. Linear speedups result in an efficiency of 1. For scalability, we define the scalability factor as efficiency measure. Using the definitions of speedup and scalability factor, for a given problem size it can be decided to add more processors based on speedup and efficiency characteristics.

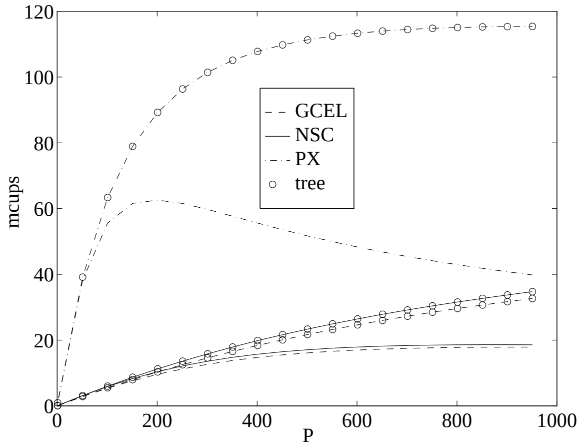
6.7 Two applications

To conclude the discussion about dataset decomposition, for two relevant applications the performance, speedup, scalability and efficiency that can be expected are presented. The two applications are *Nettalk* [93] and *Satdat*, a satellite data classification problem [89]. For the Nettalk dataset, a backpropagation neural network with 203 inputs, a varying number of hidden neurons and 26 outputs is used. The dataset and some additional information can be obtained via anonymous ftp to `ftp.idiap.ch`. The dataset contains about 20K training patterns and is contained in file `/pub/benchmarks/neural/nettalk.tar.Z`. The satellite data from Satdat contains 8047 patterns with 6 inputs (bands) and 16 classes (ground cover). For this set, a 50x50 Kohonen network is used.

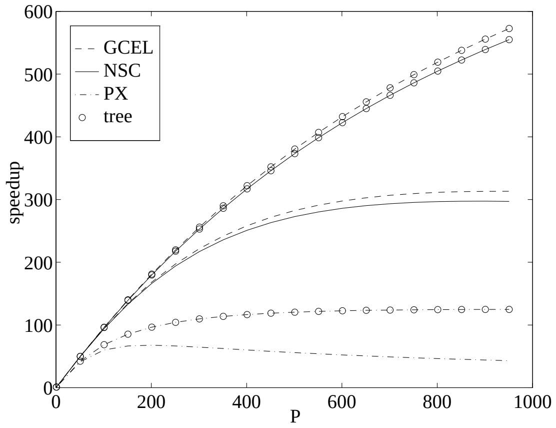
6.7.1 Dataset decomposition and Nettalk

In their paper [93], Sejnowski and Rosenberg describe a dataset and neural network architecture used to generate speech. Standard backpropagation was used. The input to the neural network represents one letter from a word to be pronounced, plus its three preceding and succeeding letters. Each letter has 29 features representing the 26 letters in English and three punctuation characters. So in total, each input pattern has 203 features. The 26 output features represent articulory features such as voicing and vowel height, and stress and syllable boundaries. In their experiments, Sejnowski and Rosenberg vary the number of hidden neurons.

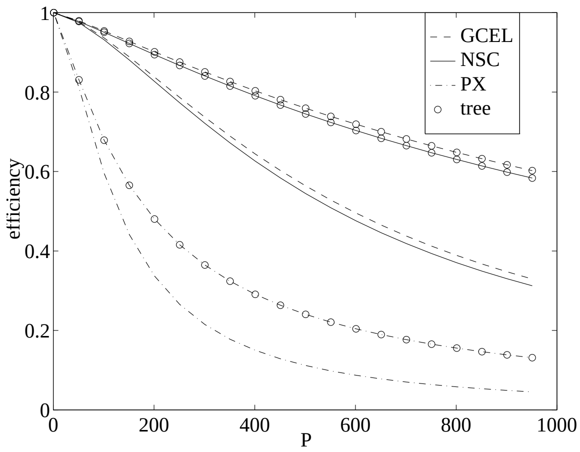
Considering the size of both dataset and neural network, Nettalk can be considered as a significant application. For the experiment depicted below, a hidden layer containing 71 neurons was chosen, so the problem size (n, w, p) is about $(300, 16K, 20K)$. Figure 6.9 depicts the expected performance, speedup and efficiency for running the application on the PX, NSC and GCEL.



(a) *Performance.*



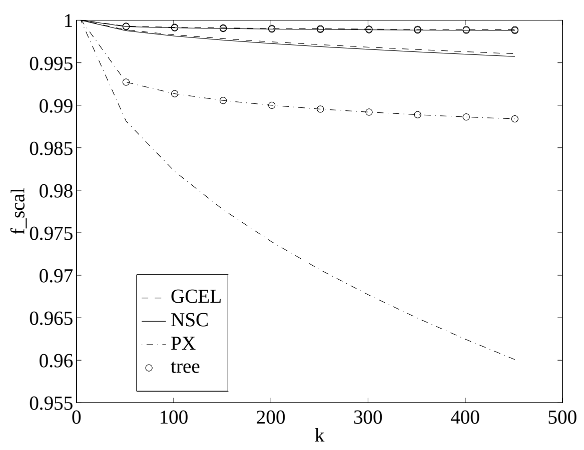
(b) *Speedups.*



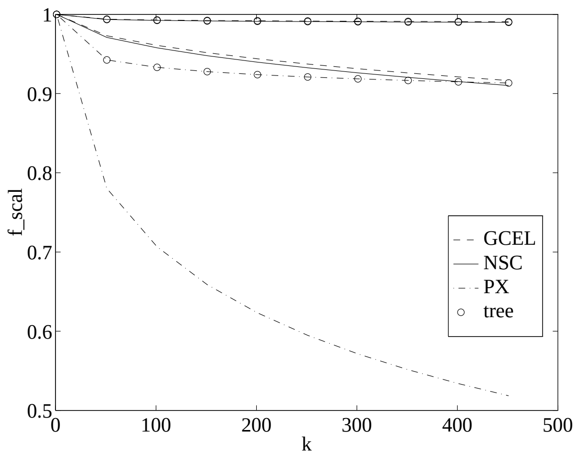
(c) *Efficiency.*

Figure 6.9: *Performance, speedup and efficiency for Nettalk.*

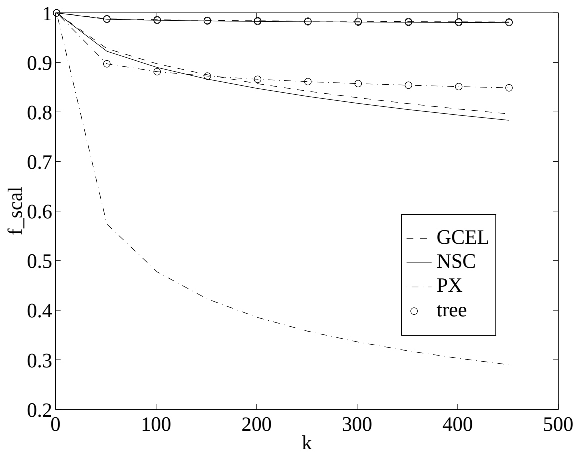
These figures are computed based on the execution time required for running one epoch, i.e. training all 20K patterns. Note that the PX is a much more powerful execution platform than the GCEL and NSC, but because of its less favorable computation/communication ratio, it is less efficient for larger processor systems. The speedup limit for PX, NSC and GCEL is respectively 194, 881 and 929 for a grid, which can be observed in Figure 6.9(b). This limit also corresponds to the plotted performance in Figure 6.9(a). For a tree, the speedup limits are much higher (respectively 986, 9579 and 10361). The expected maximum performance as predicted in Table 6.6 is not reached for large processor systems, because for large machines it only holds for much larger p. Figure 6.10 plots the expected scalability factor, for k between [0,500] and for P in {2,16,32}.



(a) *Scalability factor, P=2.*



(b) *Scalability factor, P=16.*



(c) *Scalability factor, P=32.*

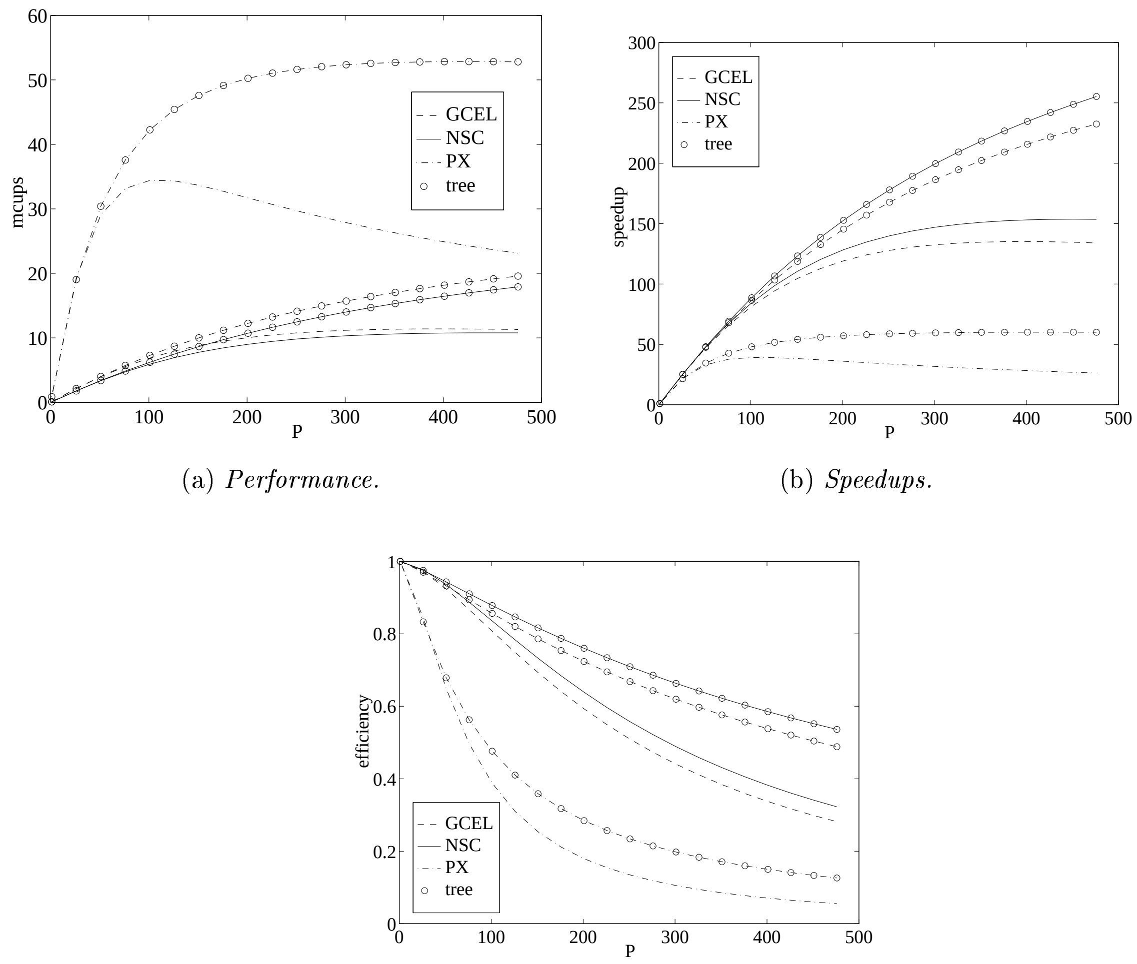
Figure 6.10: *Scalability factors for Nettalk, varying P.*

From these results it can be concluded that for a large scalability range, for any of the three execution platforms, scaling up the problem size and the number of processors results in about the same execution times; the problem is highly scalable. Especially because in practical situations k will not be very large[2], the scalability factor will be much higher than 0.9. Note that for larger P, f^{scal} decreases significantly. This is because the larger P, the more f^{scal} will approach $1/\sqrt{P}$ for a grid, and $log(P)/log(k \cdot P)$ for a tree.

[2] Even $k = 10$ can be considered high, as this means that P is increased with a factor 10.

6.7.2 Dataset decomposition and satellite data

Ron Schoenmakers, Graeme Wilkinson and Theo Schouten describe in [89] a hybrid segmentation method for classifying remotely sensed imagery. This satellite data covers the Lisbon area of Portugal, recorded with the Landsat-TM, containing 1953 lines by 1801 columns by 6 channels. Each pixel is thus represented by 6 features, and the number of ground cover classes is 16. A training set containing ground truth from a part of the image is used, in total 8047 pixels. The neural network used in [89] is a backpropagation neural network. For the purpose of this chapter, a Kohonen SOM will be used. The number of weights w in the 6x50x50 KSOM is 15K, which is somewhat smaller than the backpropagation network discussed in the previous section. Also the number of patterns for this application is smaller. Using the prediction model, the performance, speedup and efficiency are computed for this application.



(a) *Performance.*

(b) *Speedups.*

(c) *Efficiency.*

Figure 6.11: *Performance, speedup and efficiency for Satdat.*

Again, the PX offers far more performance than the GCEL and NSC. The speedup limit for PX, NSC and GCEL is respectively 110, 395, 450, (425, 2874, 3500 for a tree) which for grids can be observed in Figure 6.11(b). Note that these figures are very much similar to Figure 6.9, however, for Kohonen the GCEL is more efficient than the NSC. This is caused by the difference in computation time for the neural networks.

The scalability factors for k between [0,500] and P in $\{2,16,32\}$ are depicted in Figure 6.12. The *Satdat* problem appears highly scalable for the GCEL and NSC. For modest k, this also holds for the PX.

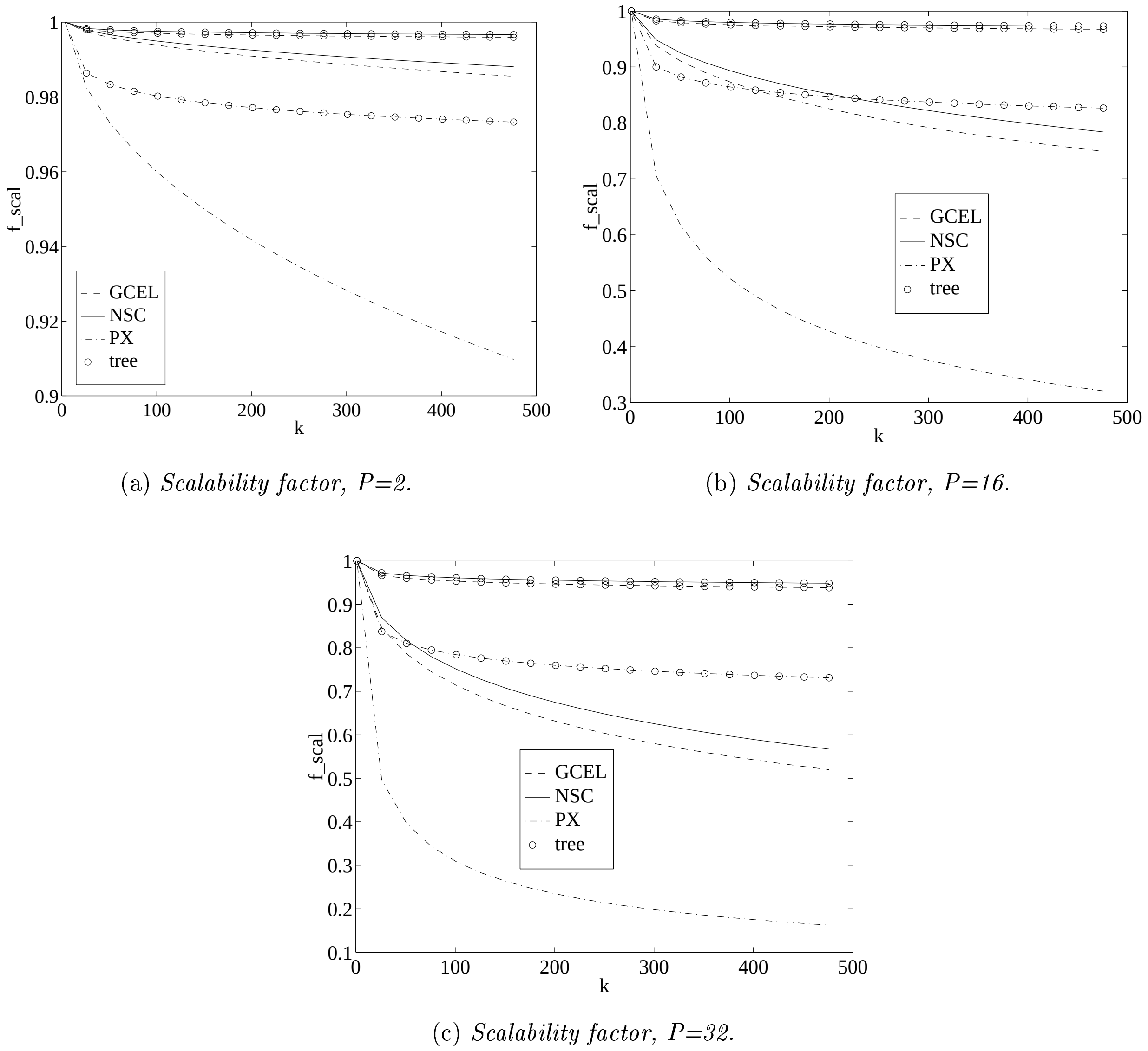


Figure 6.12: *Scalability factors for Satdat, varying P.*

When comparing these results to those achieved in Section 6.7.1, it appears that the performances are less. However, when comparing the maximum performances that can be achieved for backpropagation and Kohonen networks in Tables 6.6 and 6.14, the opposite result would be expected. The reason for this is that the efficiency of the dataset decomposition technique depends on the size of the neural network and the number of patterns. The sizes of the neural networks are about the same, but the number of patterns is higher in the first application. Therefore, its results are better.

6.8 Conclusion

Using the performance prediction model and a relatively small number of timings, it was shown how predictions can be made about performance, speedup, efficiency and scalability. By computing the point at which the derivative of the speedup becomes zero, the speedup limit could be computed. The model was validated quantitatively for Kohonen and backpropagation neural networks, resulting in accurate results within a small deviation range. It was noted that for small neural networks the predictions are less precise, but that a more elaborate modeling of kernels depending on both the number of neurons and number of weights solves this problem.

Evaluating a MIMD system can be done based on any of the predictions for performance, speedup, efficiency and scalability, or can be based on a combination of them. Depending on what question has to be answered, using the model, a decision can be made to buy more processing elements, more memory per processor or perhaps to use the available resources if extending them makes no sense. In most cases, scaling up the application is very well feasible as the scalability factors are very high. The critical consideration occurs when the problem size stays constant, because in such a case the speedup limit can be reached. For the applications considered in the previous section, which have a reasonably large problem size, it shows that current transputer systems are very well equipped for neural network simulations. As for the fast PX communication becomes a bottle neck if the number of processors increases, a faster communication network would be advantageous. Furthermore, both the GCEL as PX are 'hard-wired' in a grid topology, whereas it is shown that a tree provides a more efficient processor topology.

7

Network Decomposition

Outline

Network decomposition is defined as the collection of methods that can be used to decompose a neural network by dividing it over a number of processors. In this chapter the network decomposition of the Kohonen Self-Organizing Feature Map (KSOM) and the backpropagation neural network are discussed. It will become clear that the implementation of the Kohonen SOM is not very difficult, and therefore the focus is on the implementation aspects of backpropagation. A new communication strategy is introduced to minimize the gathering of connection updates. Furthermore, the performance prediction method described in the previous chapters will be further evaluated for network decomposition techniques.

In the previous chapter, the dataset decomposition technique is explained. If a neural network model does not accommodate epoch learning or if it does not fit on one processor, dataset decomposition cannot be exploited. The first case holds for networks that require updates of the weights immediately after being inputted with a training sample, such as Hopfield [49] or the ART networks [13]. The second case is obvious: if a neural network requires more memory for its connections or data than there is available on one processor, more processors have to be used. In these cases, the neural network has to be decomposed via another technique. Techniques where not the data, but the neural network is decomposed over the available processors are called *network decomposition* techniques. A general parallel algorithm for network-decomposed neural networks is given below in algorithm 7.1:

```
load_data();
initialize();
distribute_patterns();
while (!ready) {
  for (p=0;p<npatterns;p++) {
    compute_pattern(p);
    gather_results();
    broadcast_results();
    process_results();
  }
}
```

Algorithm 7.1: *General algorithm for network decomposition.*

This code runs on each of the nodes in the processor network. When comparing this algorithm to algorithm 6.1, it can be noted that instead of communicating after processing all patterns p/P, here for each of the p patterns communication is required. Therefore, it can be expected that network decomposition suffers more than dataset decomposition from communication overheads. In this chapter it will be shown that especially for backpropagation networks, the potential communication overheads are severe and because of this, new distributed gather routines are developed for minimizing communication. The attention in this chapter is focussed on the new gathering technique and its performance model. The concept of identifying function kernels was discussed in detail in the preceding chapters.

7.1 The backpropagation network

The network architecture of the backpropagation neural network requires that for the computation of the activation of a single unit, all activations of its incoming units are known. Each neuron in a layer is fully connected to the neurons in its input layer. The general way to decompose backpropagation networks over transputer systems is to divide each layer equally over the available processors. Consequently, all-to-all communications are necessary to provide all inputs to each neuron in a layer. Similarly, for the backward pass of the backpropagation network, all-to-all communications are required for computing each neuron's delta values. The implemented code for the network-decomposed implementations of parallel backpropagation networks discussed here is depicted in algorithm 7.2. The code represents one epoch during training:

```
for (p=0;p<npatterns;p++) {
    for (l=0;l<L-1;l++) { /* forward pass */
      broadcast_activations(l);
      compute_activations(l+1);
      gather_activations(l+1);
    }
    broadcast_activations(l);

    for (l=L-1;l>0;l--) { /* backward pass */
      get_errors_and_compute_deltas(l);
      if (l>1) {
        compute_errors(l-1);
        gather_accumulate_errors(l-1)
      }
      change_weights(l);
    }
}
```

Algorithm 7.2: *Algorithm for network-decomposed backpropagation.*

7.1.1 Implementation aspects of the forward pass

Like with the dataset decomposition techniques discussed in the previous chapter, all implementations have one master process and $P-1$ slave processes. The master loads patterns from disk, gets initial parameters and distributes this information to the slaves. Both the master and slave processes host part of the neural network. After the network initialization, each layer l contains n_l neurons, and each node in the processor network hosts (is responsible for) n_l/P neurons and $n_l \cdot n_{l-1}/P$ connections per layer $l, l>0$. The organization of neurons and weights is depicted in Figure 7.1.

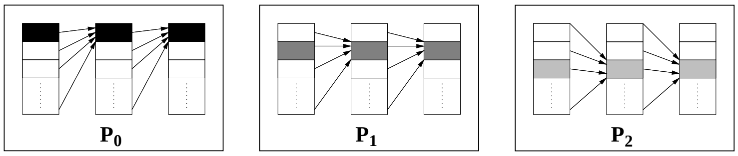


Figure 7.1: *Decomposition of layers and connections over processors, each arrow represents all connections between neurons residing on two (distinct or the same) processors.*

Note that for this decomposition method it is required that all n activations and "delta values" can be stored on each processor, which will be explained below. This will only work if the available memory is sufficient for storing $2 \cdot n$ neuron values and w/P connections on each processor. If the resources allow the storage of all patterns on each node, broadcasting the activations of layer 0 (the input layer) can be done more efficiently by just broadcasting the pattern number p_i. In the implementations discussed here, each input layer is broadcast entirely.

Consider Algorithm 7.2. In the routine `broadcast_activations()`, for layer $l = 0$, the master stores the input pattern p_i it in its input layer. For all layers $l > 0$, first a broadcast is performed of the activations in its previous layer. For $l = 1$, which is the first hidden layer, these activations are copied from the input pattern. If all activations are available, the activations of neurons in the current layer l can be computed, after which they have to be gathered and broadcast. Note that after gathering the activations of a layer l, all n_l activations are available at the master processor. After broadcasting the n_l values, all activations of this layer are available to all processors, so the local activations of the next layer can be computed. For all layers, all activations are kept in memory on each processor, which is required because of the way the backward pass is implemented. The gathering techniques discussed in Chapters 5 and 6 were tailored for simultaneously gathering *one amount of* information and accumulating it; the gather-accumulate technique. Here, when gathering the activations, there are three major differences. First, the amount of information residing on each processor may differ if $\mathrm{mod}(n_l, P) \neq 0$, i.e. there is no perfect load balance. Second, no accumulation is required. The third difference is that the structure of the information differs. Whereas with dataset decomposition all weights are gathered and accumulated, in this case the activations residing on different processors are stored in different memory locations. In Section 7.2, a technique tailored for the gathering of activations or delta values required for the backward pass is introduced.

The function kernel required for the forward pass is `compute_activations`, for which the time can be modeled as $w \cdot t_{act}$ for large networks, but (as explained in the previous chapter) for small networks it must be modeled as $n \cdot t^n_{act} + w \cdot t^w_{act}$. Using Matlab [66], for the measure pairs $A = \{(n_i, w_i)\}$ and the measured times $t = \{t_i\}$, the fitted times $f = \{t^n_{act}, t^w_{act}\}$ can be determined via $f = A \backslash t$. The time to compute n activation values is given by:

$$t_{act}(n, w) \quad = \quad n \cdot t^n_{act} + w \cdot t^w_{act} \tag{7.1}$$

and these were measured in micro seconds as:

machine	t^n_{act}	t^w_{act}
NSC	34.9	3.2
GCEL	18.3	5.9
PX	1.7	0.3

Table 7.1: *Timings for function kernel* `compute_activations`*, per neuron or weight.*

7.1.2 Implementation aspects of the backward pass

Though comprising different computations, the backward pass can be considered as the inverted forward pass. The direction of the weights is changed and the error values for each neuron j in a layer $l-1$ are computed as the in-product of the delta value of each neuron i in l and the corresponding weight values w_{ij}. By allocating memory for each layer's complete error vector, each process can compute its local contribution to the error vector as depicted in Figure 7.2:

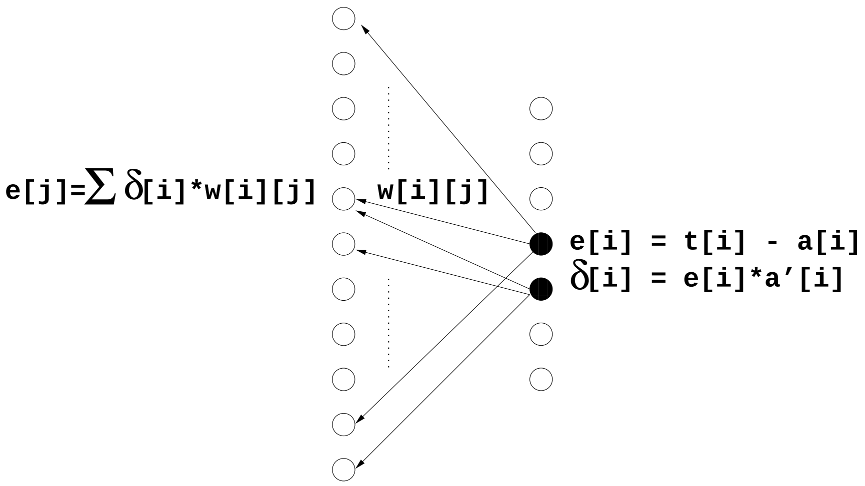


Figure 7.2: *Backpropagation of the error values by computing the contribution of all local neurons (marked black) to the errors of their input neurons.*

The algorithm given below is a more detailed description of the backward pass as introduced in algorithm 7.2:

1. The errors e_i are computed for the output layer as the difference between target and computed activation (which was kept in memory).
2. For each neuron in the output layer, its delta value δ_i is computed as the product of the error and the derivative of the activation function. This is done on all processors.
3. For each individual processor, the contribution of its local neurons to the error vector of layer $l-1$ are computed. Note that these contributions correspond with the number of weights stored locally. For each local neuron i in layer l, the contributions to each (global) neuron j in layer $l-1$ are added to the error vector of layer $l-1$ as $e_j = e_j + \delta_i \cdot w_{ij}$. This is the reason for the storage requirements of $2 \cdot n$ neuron values.
4. After computing all local contributions to the error vector of layer $l-1$, the overall vector is computed using the gather-accumulate-broadcast technique discussed before.
5. The local delta values for $l-1$ can be computed using the generalized delta rule.
6. Change the incoming weights for layer l.

7. Goto step 3 if l is not the first hidden layer.

Steps 1) and 2) are only computed for the output layer. Note that changing the weights is only allowed after computing step 3. The function kernels required for the backward pass are: `get_errors_and_compute_deltas()` (steps 1-2 or 5), `compute_errors()` (step 3), `gather_accumulate_errors()` (step 4) and `change_weights()` (step 6), costing respectively t_{gecd}, t^n_{err}, t^w_{err}, t_{gae} and t_{chw}, where t_{gae} is just $T^{GAB}(P, n_l)$.

The time to compute each neuron's delta values depends on the number of neurons n. Computing the errors in `compute_errors()` has a similar complexity as computing the activation values and is thus modeled by two factors, t^n_{err} and t^w_{err}. The time to adjust the weights depends on the number of weights w only. The function kernels were measured for different network sizes on one node of the NSC, PX and GCel. The resulting timings are listed in Table 7.2 below (micro seconds per neuron or per weight):

machine	t_{gecd}	t^n_{err}	t^w_{err}	t_{chw}
NSC	6.2	2.7	1.8	7.5
GCEL	8.1	3.5	2.6	9.8
PX	0.4	0.1	0.1	0.5

Table 7.2: *Timings for function kernels of network-decomposed backpropagation.*

The total calculation time for training one pattern of the backpropagation network is modeled by Equation (7.2). For one epoch, this time must be multiplied by the number of patterns.

$$T_{calc}(P, n, w) \quad = \quad \frac{1}{P} \cdot (n \cdot (t^n_{act} + t_{gecd} + t^n_{err}) + w \cdot (t^w_{act} + t^w_{err} + t_{chw}) \qquad (7.2)$$

The calculation times for computing Equation (7.2) were measured on the three transputer platforms. From Figure 7.3 it can be observed that for larger neural networks, the deviation between measured and predicted times is very small. Furthermore, even for small networks, the deviations are within 8%.

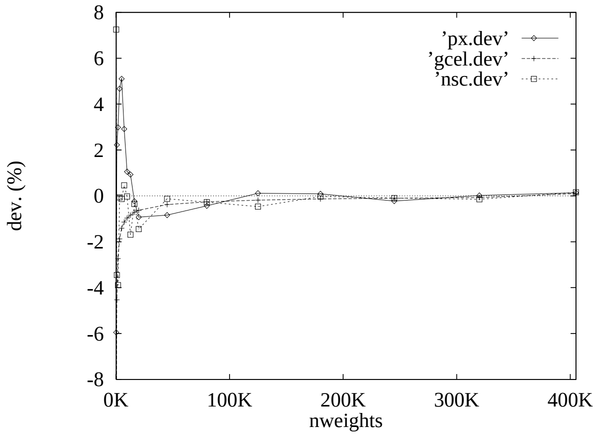


Figure 7.3: *Deviations in % for Equation (7.2) and all three machines for different network sizes.*

7.2 A new gathering technique

In this section a new gathering technique is introduced which is tailored for gathering an amount of information which is distributed over a processor network. This is different from the gather-accumulate-broadcast technique discussed before. In the latter technique, several different instances of the same information are located on distinct processors. After gathering, accumulating and broadcasting, the updated information is available to all processors. In the new technique, the information to be updated can be considered as a vector of n elements, where each processor holds n/P elements. After computing n/P new element values, the goal is to make all new values available to all processors. Two techniques to perform this operation are compared below.

7.2.1 The store-and-forward technique for grids

In a first attempt, the problem was attacked as depicted in Figure 7.4. In a horizontal phase, after $W-1$ communication steps, all "row" information is available at the left most column of a grid. Similarly in the vertical phase, after $H-1$ steps, all n elements are available at the root processor.

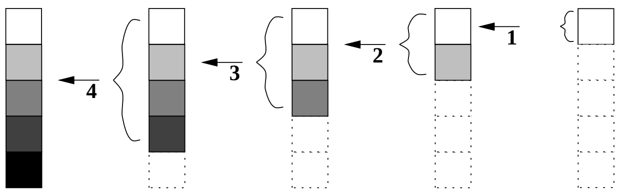


Figure 7.4: *Gathering using the store-and-forward technique.*

For a $\sqrt{P} * \sqrt{P}$ grid, the total amount of transmitted elements N_h for the horizontal phase is given in Equation (7.3). The amount of transmitted elements gathered at the left most column is $\sqrt{P} \cdot n/P$. This means that for the vertical phase $N_v = \sqrt{P} \cdot N_h$, so the total amount N_{total} can be given by Equation (7.4):

$$
\begin{aligned}
N_h &= \frac{n}{P} + 2 \cdot \frac{n}{P} + 3 \cdot \frac{n}{P} + \ldots + (\sqrt{P} - 1) \cdot \frac{n}{P} \\
&= \frac{n}{P} \cdot \frac{\sqrt{P} \cdot (\sqrt{P} - 1)}{2} \qquad (7.3) \\
N_v &= \frac{n}{P} \cdot \sqrt{P} \cdot \frac{\sqrt{P} \cdot (\sqrt{P} - 1)}{2} \\
N_{total} &= \frac{n}{2 \cdot P} \cdot \sqrt{P} \cdot (\sqrt{P} + 1) \cdot (\sqrt{P} - 1) \\
&= \frac{n}{2 \cdot P} \cdot \sqrt{P} \cdot (P - 1) \qquad (7.4)
\end{aligned}
$$

This technique is called store-and-forward, because only if a processor has received all information from neighbors situated more "inside" the processor network, it sends all gathered information so-far to its parent processor. For example for the horizontal phase of this technique, all processors in the left-most column have to wait $\sqrt{P} - 2$ steps before they may receive information.

7.2.2 The pipeline techniques for grids

The new *distributed* gathering technique introduced here makes better use of the available resources. Each processor first sends its information to its neighbors, before receiving information from "deeper laying" processors. As can be observed in Figure 7.5, also after $W-1$ steps, all "row" information is available at the left-most processors. During each step, 2 communications are performed, i.e., sending to a neighbor and subsequently receiving from a deeper node. However, on the transputer, inter-link communication can be done in parallel and a write operation does not have to wait for receipt. Therefore, the two communication steps can be done in one pass, where the setup time is ignored.

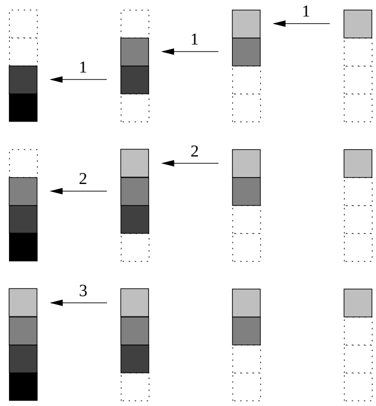


Figure 7.5: *Gathering using the pipeline technique.*

At step i, each processor P_j has gathered information of processors $P_{j+1} \ldots P_{j+i}$. This means for a $\sqrt{P} * \sqrt{P}$ grid, that after step $\sqrt{P} - 1$, all left-most processors have gathered all information in a row. Similar considerations can be made for the vertical phase, so the total amount of elements transmitted can be derived as specified below:

$$
\begin{aligned}
N_h &= \frac{n}{P} \cdot (\sqrt{P} - 1) \\
N_v &= \frac{n}{P} \cdot \sqrt{P} \cdot (\sqrt{P} - 1) \\
N_{total} &= \frac{n}{P} \cdot (1 + \sqrt{P}) \cdot (\sqrt{P} - 1) \\
&= \frac{n}{P} \cdot (P - 1)
\end{aligned} \tag{7.5}
$$

From Equations (7.4) and (7.5) it can be deduced that the pipelined technique performs significantly ($\sqrt{P}/2$) faster than the store-and-forward technique. However, the pipeline technique is not capable of speeding up the GAB techniques described in Chapters 5 and 6. At first instance this seems to be the case, because in the GAB techniques also store-and-forward principles are used. However, when gathering one amount of information s residing at each processor, using the store-and-forward technique as well as the pipeline technique, the traffic would amount to:

$$
\begin{aligned}
N_h &= s \cdot (\sqrt{P} - 1) \\
N_v &= s \cdot (\sqrt{P} - 1) \\
N_{tot} &= s \cdot (2 \cdot \sqrt{P} - 1)
\end{aligned}
$$

Below, an algorithm for gathering the activations of a layer l using the pipeline technique is given.

```
int offst = lindex[pid][l];
int n = nneurons_in[l]/W/H; /* WxH grid */
int i;

/* horizontal phase */
if (has_left)
  snd(left,&activations[l][offst],n);
for (i=0;i<nodesright;i++) {
  offset += n;
  rec(right,&activations[l][offst],n);
  if (has_left)
    snd(left,&activations[l][offst],n);
}
```

(a) *Initialization and horizontal phase.*

```
/* vertical phase */
offst = lindex[pid][l];
if (!has_left) { /* the left most column */
if (has_top)
    snd(top,&activations[l][offst],n*W);
  for (i=0;i<nodesdown;i++) {
    offset += n*W;
    rec(down,&activations[l][offst],n*W);
    if (has_top)
      snd(top,&activations[l][offst],n*W);
  }
}
```

(b) *Vertical phase.*

Algorithm 7.3: *Algorithm for distributed gathering.*

In algorithm 7.3, two simplifications are made. First, an equal load balance is assumed, where each processor has the same amount of n_l/P activations, and n_l/P is an integer. Second, the distribution of a layer over processors is done as depicted in Figure 7.6, which allows for the simple administration of the used datastructures. Each processor "knows" that the index to data from its right neighbor is just its own offset plus n_l/P. In the actual implementations, during an initial decomposition phase, all processors get a datastructure containing information about the number of neurons on each other processor. Using this datastructure, at each communication step it is known how many information has to be communicated, and where it has to be stored.

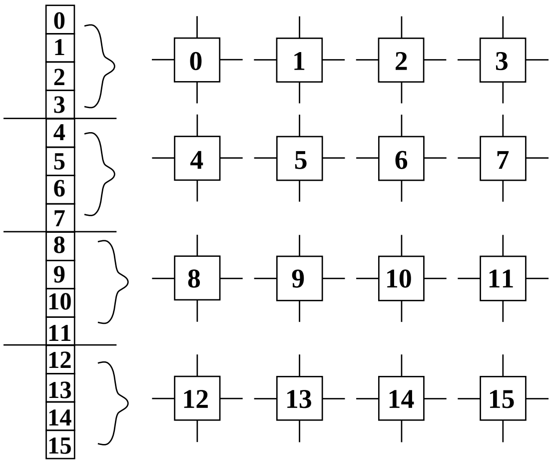


Figure 7.6: *Gathering using the pipeline technique.*

7.3 Backpropagation communication costs

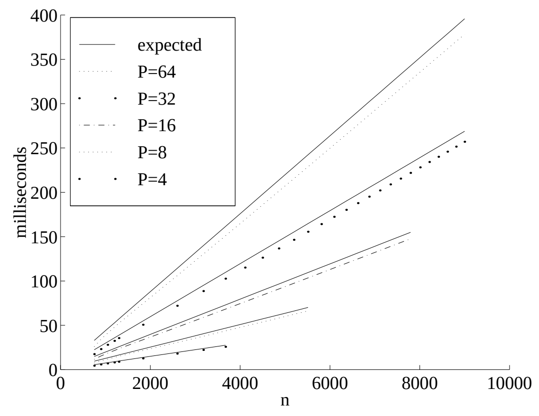
For the forward pass of backpropagation, the time for broadcasting and subsequently distributed gathering the activations for each layer $l \in [0 \cdots L-1]$ can be estimated as:

$$\begin{aligned} T_{grid}^{DGB}(P,n,w) &= \sum_{l=0}^{L-1} T_{grid}^{B}(P,n_l) + \sum_{l=1}^{L-1} T_{grid}^{DG}(P,n_l) \\ &= (n_0+n_1+n_2)\cdot(2\cdot\sqrt{P}-2)\cdot t_{comm} + \\ &\quad \frac{n_1+n_2}{P}\cdot(P-1)\cdot t_{comm} \\ &= t_{comm}\cdot\left(n\cdot(2\cdot\sqrt{P}-2)+\frac{n-n_0}{P}\cdot(P-1)\right) \end{aligned} \tag{7.6}$$

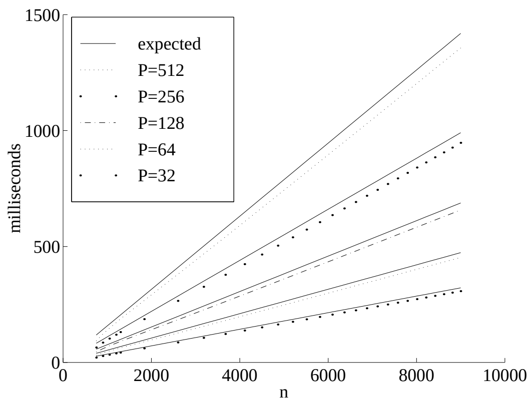
For the backward pass, as explained in Section 7.1.2, it is assumed that each processor stores all activations computed during the forward pass. In this way, the required communications can be limited to gathering, accumulating and broadcasting the error vector:

$$T_{grid}^{GAB}(P,n-n_0)$$

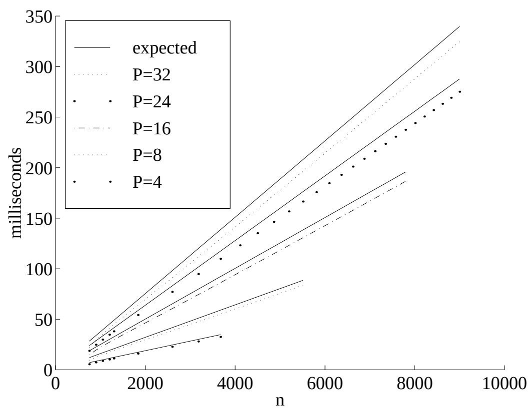
The time required for the forward pass was measured on each of the three platforms NSC, GCEL and PX. Measurements were carried out using the "ping-pong" technique, i.e. the root processor starts the timer, first broadcasts activations from a layer l and stops the timer after receiving all activations from $l-1$.



(a) *nsc*



(b) *gcel*



(c) *power*

Figure 7.7: *Measured and expected communication times for NSC, GCEL and PX.*

As can be observed in Figure 7.7, deviations are relatively small. For all three platforms, deviations are within 4%-15%. The total execution time is is derived from Equations (7.2), (7.6) and $T^{GAB}_{grid}(P, n-n_0)$ as:

$$\begin{aligned} T^{ex}_{grid}(P,n,w) &= T_{calc}(P,n,w) + T^{DGB}_{grid}(P,n,w) + T^{GAB}_{grid}(P,w) \\ &= T_{calc}(P,n,w) + t_{comm} \cdot (n \cdot (2 \cdot \sqrt{P} - 2) + \frac{n-n_0}{P} \cdot (P-1)) + \\ & \quad (2 \cdot t_{comm} + t_{acc}) \cdot (n - n_0) \cdot (2 \cdot \sqrt{P} - 2) \qquad (7.7) \end{aligned}$$

7.4 Backpropagation on a tree

In [110] and in Chapter 5, it was concluded that the tree topology offers an efficient way of performing gather-accumulate-broadcast operations. The major point in this technique is that accumulations are performed in parallel during communication. From algorithm 7.2 it becomes clear that the GAB technique is used for accumulating the neuron error vectors during the backward pass. During the forward pass however, no accumulation is required. In this section, the communication time for the network decomposed backpropagation model on trees is derived. As the time required for the backward pass is $T_{tree}^{GAB}(n - n_0)$, the attention is focussed on the forward pass.

7.4.1 Communication time for the forward pass

Again, an equal load balance is assumed, so each processor hosts ${}^{n_l}/_{p}$ neurons per layer l. Two techniques are compared for gathering distributed information: the store-and-forward technique and the pipelined technique. In the first technique, each processor on a certain level of the tree first receives all activation values of its sub-trees, and then transmits these together with its own values to its parent. For a tree of depth 2 (i.e., the root with three sub-nodes, the number of communications is $3 \cdot 1$. For a tree of depth 3, this number is $3 \cdot (3+1)$, etcetera. For a depth of d, the number of required communications for gathering distributed information is as depicted in Figure 7.8:

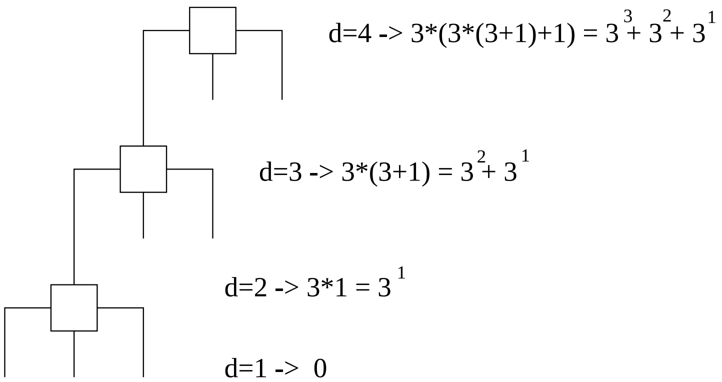


Figure 7.8: *Number of communications for gathering via the store-and-forward technique.*

The total time for gathering and subsequently broadcasting ${}^{n_l}/_{P}$ neurons can be derived as Equation (7.8):

$$
\begin{aligned}
T_{tree}^{DGB} &= t_{comm} \cdot \frac{n_l}{P} \cdot (3^1 + 3^2 + \cdots + 3^{d-1}) + T_{tree}^{B}(P, n_l) \\
&= t_{comm} \cdot \frac{n_l}{P} \cdot \sum_{i=1}^{d-1} 3^i + T_{tree}^{B}(P, n_l) \\
&= t_{comm} \cdot \frac{n_l}{P} \cdot (\frac{3^d - 1}{2} - 1) + T_{tree}^{B}(P, n_l) \qquad (7.8)
\end{aligned}
$$

Using the pipelined technique, each processor sends its local information to its parent node, and subsequently receives and transmits the information from each of the nodes in its three sub-trees. It will be deduced below through inductive proof, that the required number of communications for a number of nodes $N(d)$ for a depth d equals $N(d) - 1$.

d=1 There are zero leaves, so the number of communications required is zero.

d=2 There are four nodes and three leaves, so the number of communications required is three.

d=k Each of the three nodes down the root has a depth $k - 1$. Each of these has gathered all information in its subtree in $N(k-1)-1$ steps (through induction). Furthermore, the root has already received $N(k - 1) - 1$ packets from one of the three nodes (say its left-down neighbor). This means that another $1 + 2 * N(k - 1)$ communications are required, which in total amounts to $3 * N(k - 1) = N(k) - 1$.

Based on these observations, it is concluded that the total communication time for a tree of depth d equals Equation (7.9), which is equal to Equation (7.8). This involves that there is no difference between the store-and-forward and pipelined techniques for trees.

$$\begin{aligned} T^{DG}_{tree} &= t_{comm} \cdot \frac{n_l}{P} \cdot \sum_{i=0}^{d-1} 3^i \\ &= t_{comm} \cdot \frac{n_l}{P} \cdot (\frac{3^d}{2} - 1) \end{aligned} \tag{7.9}$$

The total communication time required for the forward pass is given below:

$$\begin{aligned} T^{DGB}_{tree}(P, n, w) &= \sum_{l=0}^{L-1} T^{B}_{tree}(P, n_l) + \sum_{l=1}^{L-1} T^{DG}_{tree}(P, n_l) \\ &= (n_0 + n_1 + n_2) \cdot 3 \cdot depth(P) \cdot t_{comm} + \\ &\quad \frac{n_1 + n_2}{P} \cdot (\frac{3^d}{2} - 1) \cdot t_{comm} \\ &= t_{comm} \cdot (n \cdot 3 \cdot depth(P) + \frac{n - n_0}{P} \cdot (\frac{3^d}{2} - 1)) \end{aligned} \tag{7.10}$$

The total execution time for running network decomposed backpropagation networks on a tree is derived from Equations (7.2), (7.10) and $T^{GAB}_{tree}(P, n - n_0)$ as:

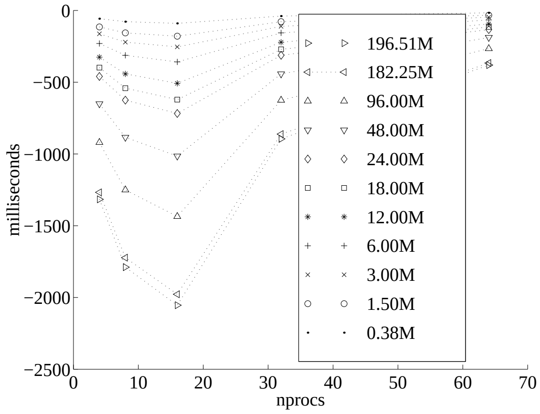
$$\begin{aligned} T^{ex}_{tree}(P, n, w) &= T_{calc}(P, n, w) + T^{DGB}_{tree}(P, n, w) + T^{GAB}_{tree}(P, w) \\ &= T_{calc}(P, n, w) + t_{comm} \cdot (n \cdot 3 \cdot depth(P) + \frac{n - n_0}{P} \cdot (\frac{3^d}{2} - 1)) + \\ &\quad (2 \cdot t_{comm} + t_{acc}) \cdot (n - n_0) \cdot 3 \cdot depth(P) \end{aligned} \tag{7.11}$$

7.5 A comparison between transputer grids and trees

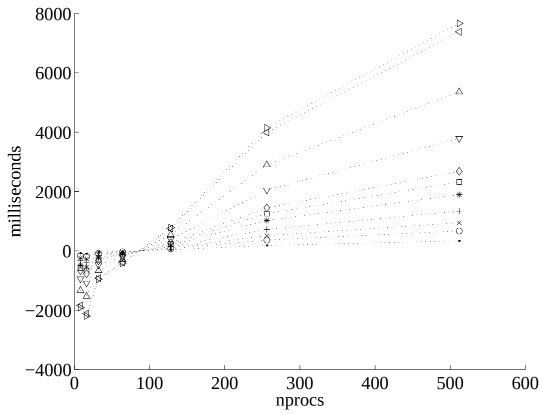
To compare whether a tree or grid topology is more suited for network decomposed back-propagation networks, the difference between Equations (7.7) and (7.11) is considered.

$$\begin{aligned} T^{ex}_{grid}(P,n,w) - T^{ex}_{tree}(P,n,w) &= t_{comm} \cdot (n \cdot (2 \cdot \sqrt{P} - 2) + \frac{n - n_0}{P} \cdot (P - 1)) + \\ & (2 \cdot t_{comm} + t_{acc}) \cdot w \cdot (2 \cdot \sqrt{P} - 2) - \\ & t_{comm} \cdot (n \cdot 3 \cdot depth(P) + \frac{n - n_0}{P} \cdot (\frac{3^d}{2} - 1)) - \\ & (2 \cdot t_{comm} + t_{acc}) \cdot (n - n_0) \cdot 3 \cdot depth(P) \qquad (7.12) \end{aligned}$$

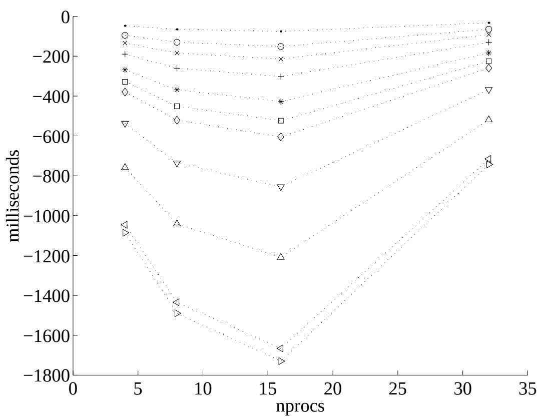
Below, the expected difference between the communication times required for grids and trees are depicted.



(a) *Differences for NSC*



(b) *Differences for GCEL*



(c) *Differences for PX*

Figure 7.9: *Expected difference in communication times between grids and trees.*

In Figure 7.9, it can be observed that for a larger number of processors, a tree topology is more efficient than a grid. For a smaller number of processors (< 80), a grid shows to be more efficient. This effect can also be determined by considering Equation (7.12), which can be rewritten to:

$$\begin{aligned}
&t_{comm} \cdot n \cdot (2 \cdot \sqrt{P} - 2 - 3 \cdot depth(P)) + \\
&t_{comm} \cdot \frac{n - n_0}{P}(P - \frac{3^d}{2}) + \\
&(n - n_0) \cdot (2 \cdot t_{comm} + t_{acc}) \cdot (2 \cdot \sqrt{P} - 2 - 3 \cdot depth(P))
\end{aligned}$$

If the depth of a ternary tree is estimated as ${}^3log(P)$, the term $2 \cdot \sqrt{P} - 3 \cdot {}^3 log(P)$ rules Equation (7.12) for larger P. In the discussions below, only results for a grid are considered. This is because of the following considerations:

1. Current transputer systems are all hard-wired in a grid topology; only virtual (and thus non-efficient) tree-topologies can be made.

2. Developments in computer hardware have shown that large MIMD parallel processor systems in general cannot keep up with the pace with which processors are accelerated. This was in particular true for the T8xx transputer systems, which were in fact already out-dated by the time they were released.

3. On the other hand, smaller systems are now being released, like the PowerXPlorer. These machines contain nodes with more state-of-the-art performance.

4. For decomposition strategies like network decomposition which require relatively much communication overheads, such machines provide a more efficient platform than large machines containing many, less-powerful nodes.

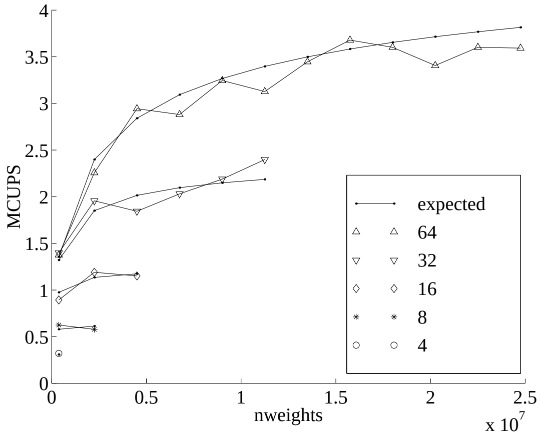
The latter argument will be explained in more detail below. It will be shown that especially for network decomposition, smaller systems offer a higher efficiency than larger ones.

7.6 Performance

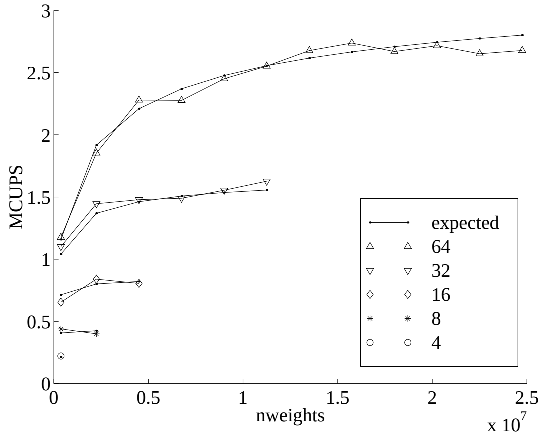
The performance for network backpropagation networks decomposed via network decomposition is:

$$\mathcal{P}(P, n, w) \;=\; \frac{w}{T^{ex}_{grid}(P, n, w)} \text{ MCUPS}$$

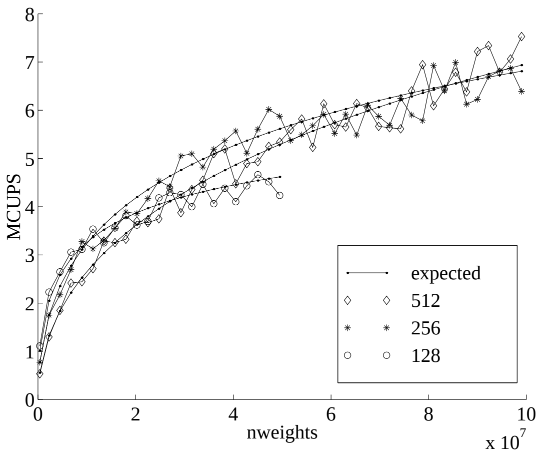
The measured and expected performance are depicted in Figure 7.10:



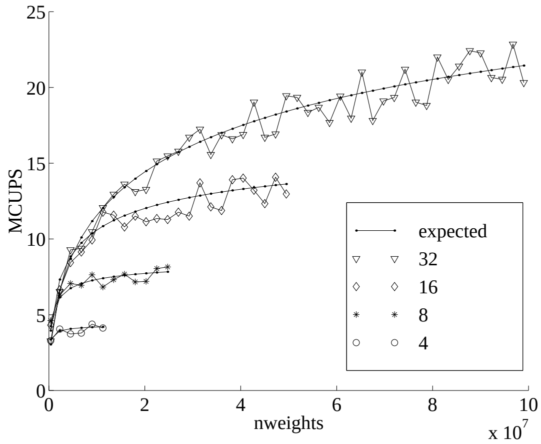
(a) *Performance for NSC*



(b) *Performance for small GCEL*



(c) *Performance for large GCEL*



(d) *Performance for PX*

Figure 7.10: *Expected and measured performance in* MCUPS.

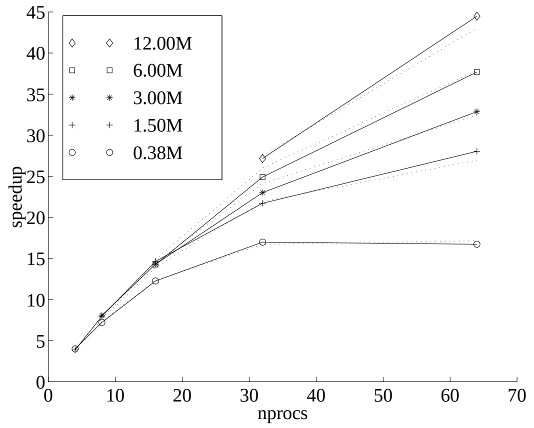
Compared to the performance achieved with the dataset decomposition techniques explained in the previous chapter, these performances are significantly lower. This was already expected and is due to the increased communication overheads. Note that for smaller neural networks, this effect involves that the performance is lower on larger grids. In particular consider Figure 7.10(c), where 512 processors only reach a higher performance than 128 processors for problems larger than 85M weights. The deviations between measured and expected performance are all within 15%. In Chapter 6, the maximum performance that can be achieved was computed by taking the limit for the problem size. The problem size can be enlarged by increasing the number of patterns or by increasing the size of the neural network. In this case, the execution time increases linearly with the number of patterns, and the size of the neural network is limited by the amount of available memory. Therefore, for network decomposed backpropagation networks, the maximum performance cannot be computed.

7.7 Speedup, scalability and efficiency

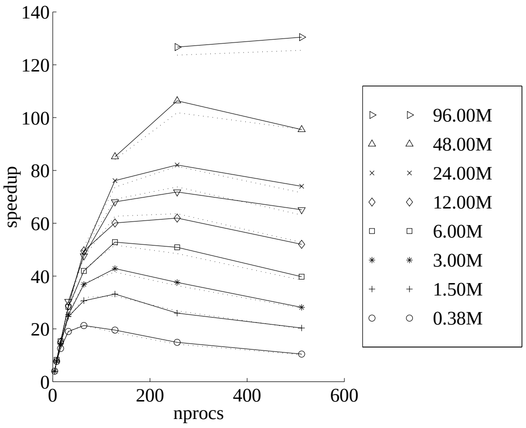
The (fixed-size) speedup is computed as:

$$\mathcal{S}(P,n,w) = \frac{T^{ex}_{grid}(1,n,w)}{T^{ex}_{grid}(P,n,w)} = \frac{T_{calc}(1,n,w)}{T^{ex}_{grid}(P,n,w)} \tag{7.13}$$

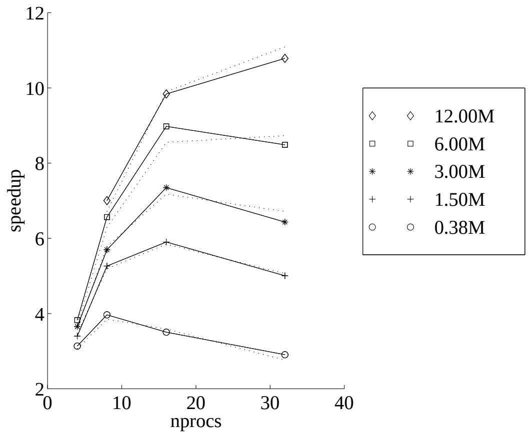
Below, the measured and expected speedup are depicted.



(a) *Speedup for NSC*



(b) *Speedup for GCEL*



(c) *Speedup for PX*

Figure 7.11: *Expected and measured speedup in* MCUPS.

When considering the fixed size speedup, it is noted that:

1. The achieved speedups are lower than those achieved with dataset decomposition.
2. For smaller neural networks, the speedup limit is reached relatively soon.

The speedup limit for a given neural network can be computed by computing the derivative of Equation (7.13) and solving it equal zero. Let:

$$\begin{aligned}
a &= T_{calc}(1,n,w) \\
b &= T_{comm} * n * 2 \\
c &= (n - n_0) * t_{comm} \\
d &= (2 * t_{comm} + t_{acc}) * 2 * (n - n0) \\
\mathcal{S}(P,n,w) &= \frac{T_{calc}(1,n,w)}{T^{ex}_{grid}(P,n,w)} \\
\left[\frac{T_{calc}(1,n,w)}{T^{ex}_{grid}(P,n,w)}\right]' &= 0 \\
\Longleftrightarrow \left[\frac{a}{\frac{a}{P} + b * \sqrt{P} - b + c - \frac{c}{P} + d * \sqrt{P} - d}\right]' &= 0 \\
\Longleftrightarrow -a * \left(\frac{c-a}{P^2} + \frac{b+d}{\sqrt{P}}\right) &= 0 \\
\Longleftrightarrow c - a + \frac{(b+d)}{2} * P * \sqrt{P} &= 0 \\
\Longleftrightarrow P &= \left(\frac{2 \cdot (a-c)}{b+d}\right)^{\frac{2}{3}} \qquad (7.14)
\end{aligned}$$

The following table depicts the computed speedup limit and corresponding maximal speedup following Equation (7.14). In Figure 7.11, it can be observed that these numbers match the points at which the speedup drops.

	NSC		**GCEL**		**PX**	
Mweights	P	$\mathcal{S}(P)$	P	$\mathcal{S}(P)$	P	$\mathcal{S}(P)$
0.38M	48	(18)	59	(22)	9	(4)
1.50M	76	(27)	94	(34)	15	(6)
3.00M	95	(34)	118	(42)	19	(7)
6.00M	120	(43)	148	(52)	24	(9)
12.00M	151	(53)	187	(66)	30	(11)
18.00M	173	(61)	214	(75)	34	(13)
24.00M	190	(67)	235	(82)	38	(14)
48.00M	240	(84)	297	(103)	47	(17)
60.00M	258	(90)	320	(111)	51	(19)
96.00M	302	(105)	374	(129)	60	(22)

Table 7.3: *Speedup limits following Equation (7.14).*

In the tables depicted below, some computed scalability factors for network decomposed backpropagation networks are given, for each of the three target platforms. The scalability factor is computed as:

$$f^{scal}(k, P, (n, w)) \quad = \quad \frac{T(P, n, w)}{T(k \cdot P, k \cdot (n, w))}$$

k	(.4M,P=1)	(1.6M,P=4)	(.4M,P=8)	(3M,P=8)
1	1.00 (1)	1.00	1.00	1.00
2	1.00 (2)	0.98	0.90	0.96
4	0.99 (4)	0.95	0.75	0.89
8	0.98 (7)	0.87	0.56	0.77
16	0.93 (15)	0.76		
32	0.87 (28)			
64	0.75 (48)			

Table 7.4: *Scalability factors for NSC (scalability in parentheses).*

k	(.4M,P=1)	(1.6M,P=4)	(.4M,P=8)	(3M,P=8)
1	1.00 (1)	1.00	1.00	1.00
2	1.00 (1)	0.99	0.93	0.92
4	0.99 (4)	0.96	0.80	0.82
8	0.98 (7)	0.90	0.63	0.67
16	0.95 (15)	0.81	0.43	0.50
32	0.90 (29)	0.67	0.27	0.32
64	0.80 (51)	0.49	0.15	
128	0.66 (84)	0.32		
256	0.49 (125)			
512	0.32 (164)			

Table 7.5: *Scalability factors for GCEL (scalability in parentheses).*

k	(.4M,P=1)	(1.6M,P=4)	(.4M,P=8)	(3M,P=8)
1	1.00 (1)	1.00	1.00	1.00
2	0.96 (2)	0.84	0.60	0.73
4	0.87 (3)	0.62	0.33	0.48
8	0.73 (6)	0.40		
16	0.54 (9)			
32	0.35 (11)			

Table 7.6: *Scalability factors for PX (scalability in parentheses).*

Similar to the dataset decomposition technique described in Chapter 6, it shows here that for larger k, the scalability factor drops. Also note that still no scalability limit is found.

7.8 Expected results for Nettalk

The expected performance, speedup and efficiency for the application Nettalk described in the previous chapter are computed here.

	Performance (MCUPS)			**Speedup**			**Efficiency**		
P	NSC	GCEL	PX	NSC	GCEL	PX	NSC	GCEL	PX
1	0.08	0.05	1.06	1.00	1.00	1.00	1.00	1.00	1.00
2	0.15	0.10	1.73	1.97	1.97	1.63	0.98	0.98	0.81
4	0.28	0.20	2.06	3.70	3.77	1.93	0.93	0.94	0.48
8	0.47	0.35	1.77	6.21	6.54	1.66	0.78	0.82	0.21
16	0.62	0.49	1.28	8.26	9.28	1.20	0.52	0.58	0.07
32	0.62	0.53	0.88	8.24	9.91	0.82	0.26	0.31	0.03
64	0.50	0.44	0.60	6.64	8.34	0.56	0.10	0.13	0.01
128	0.36	0.33	0.41	4.86	6.22	0.38	0.04	0.05	0.00
256	0.26	0.23	0.28	3.43	4.42	0.27	0.01	0.02	0.00
512	0.18	0.16	0.20	2.40	3.11	0.18	0.00	0.01	0.00

Table 7.7: *Performance, speedup and efficiency Nettalk*

Note that the performance appears to be very low compared with the results depicted in Figure 6.9. The reasons for this result are, again, the communication overheads. With dataset decomposition only after computing the weight changes for 20K patterns, communication has to take place. Here, the overheads are so severe, that for this application only small processor networks are suited. For larger systems, the performance drops. Using Equation (7.14), the number of processors at which this occurs can be computed. The maximal speedup for respectively the NSC, GCEL and PX is found at respectively $P = 22$, $P = 26$ and $P = 4$:

	P	speedup=$\mathcal{S}(P)$	performance=$\mathcal{P}(P)$(MCUPS)	efficiency=$\mathcal{E}$(P)
NSC	22	8.6	0.64	0.39
GCEL	26	10.1	0.53	0.39
PX	4	2.0	2.00	0.5

Table 7.8: *Maximum speedup and corresponding performance and efficiency for Nettalk.*

7.9 The Kohonen neural network

In Section 4.2.2.1, the Kohonen Self-Organizing feature map (KSOM) algorithm is explained. Here, the parallel implementation of the KSOM via network decomposition is discussed. For network decomposition, when distributing the neurons of the KSOM, two requirements must be met [7, 104]:

1. Divide all neurons equally over the available processors, where each neuron "hosts" its weights locally. This requirement assures that during the recall phase (finding the winning neuron), the computational load is well-balanced.

2. Make sure that neighboring neurons are placed at distinct processors. This assures that during the training phase (changing the weights of neurons in a certain neighborhood of the winning neuron), the load is well-balanced.

The architecture of the KSOM is a grid with width Kw and height Kh, which has to be decomposed over P processors. A simple algorithm which ensures that no neighboring neurons lay on the same processor and equally divides neurons over the available processors is a *round-robin decomposer*. On each processor, an array `global_pos` is available pointing at positions in the Kohonen map. The distribution is performed by the master process, and it is assumed that it can send information to a particular processor using a `send` command:

```
void distribute_network (int Kw, int Kh, int P)
{
     int i,n,p;

     for (i=0;i<Kw*Kh;i++) {
          p = i%P;
          global_pos[p][nneurons_on[p]] += i;
          nlocal_neurons[p] += 1;
     }
     for (i=0;i<P;i++) {
          send(i,(char *)&nlocal_neurons[i],sizeof(int));
          send(i,(char *)global_pos[i],nlocal_neurons[i]*sizeof(int));
     }
}
```

7.9.1 Finding the winning neuron

After this initial decomposition phase, communication is only required for the distribution of input patterns (implemented by a `broadcast` operation), and for the distributed computation of the winning neuron. The implementation of the latter computation is very similar to the GAB technique described before. Instead of accumulating weights, now only two values are communicated: the index of the winning neuron and its distance to the input pattern. On each processor, the local "winner" and its distance to the input is computed.

Using the distributed gathering technique, the global winner is determined. Below, this algorithm is given for a tree and grid processor topology.

```
void gather_winner (int *winner, double *mydist)
{
  int w;
  double d;

  if (has_right) {
    rec (EAST,(char *) &w,sizeof(int));
    rec (EAST,(char *) &d,sizeof(double));
    if (*mydist>d) {
      *mydist = d;
      *winner = w;
    }
  }
  if (!has_left) { /* leftmost row */
    if (has_down) {
      rec (SOUTH,(char *) &w,sizeof(int));
      rec (SOUTH,(char *) &d,sizeof(double));
      if (*mydist>d) {
        *mydist = d;
        *winner = w;
      }
    }
    if (has_top) {
      snd (NORTH,(char *) winner,sizeof(int));
      snd (NORTH,(char *) mydist,sizeof(double));
    }
  }
  else {
      snd (WEST,(char *) winner,sizeof(int));
      snd (WEST,(char *) mydist,sizeof(double));
  }
  broadcast ((char *) winner, sizeof(int));
  broadcast ((char *) mydist, sizeof(double));
}
```

(a) *Finding the winner on grids.*

```
void gather_winner (int *winner, double *mydist)
{
  int w;
  double d;

  if (has_left) {
    rec (LEFT,(char *) &w,sizeof(int));
    rec (LEFT,(char *) &d,sizeof(double));
    if (*mydist>d) {
      *mydist = d;
      *winner = w;
    }
  }
  if (has_down) {
    rec (DOWN,(char *) &w,sizeof(int));
    rec (DOWN,(char *) &d,sizeof(double));
    if (*mydist>d) {
      *mydist = d;
      *winner = w;
    }
  }
  if (has_right) {
    rec (RIGHT,(char *) &w,sizeof(int));
    rec (RIGHT,(char *) &d,sizeof(double));
    if (*mydist>d) {
      *mydist = d;
      *winner = w;
    }
  }
  if (has_top) {
    snd (TOP,(char *) winner,sizeof(int));
    snd (TOP,(char *) mydist,sizeof(double));
  }
  broadcast ((char *) winner, sizeof(int));
  broadcast ((char *) mydist, sizeof(double));
}
```

(b) *Finding the winner on trees.*

Algorithm 7.4: *Finding the winner on grids and trees.*

After gathering the winner, on each processor it is known which neuron is the winning one. Based on its index, the position in the KSOM can be computed as $(idx\%Kw, idx/Kw)$, and for all neurons on a processor, it can be decided whether their weights have to be updated by considering whether they are in the neighborhood of the winner. The network-decomposed KSOM algorithm is depicted below:

```
void ksom ()
{
  int i;

  distribute_network();
  for (i=0;i<p;i++) {
    broadcast ((char *)pattern,ninputs*sizeof(float));
    winner = FindWinner(pattern);
    ChangeWeights(pattern,winner);
  }
}

void ChangeWeights (float *pat, int w)
{
  int i,j;
  register float *iptr;

  for (i=0;i<nlocal_neurons;i++) {
    gauss_value = in_neighbourhood(globalpos[i],w);
    if (gauss_value>0.0)
      for (j=0;j<ninputs;j++)
        weights[i][j] += lrate*gauss_value*(pat[j]-weights[i][j]);
    }
}
```

Algorithm 7.5: *General algorithm for Kohonen network decomposition.*

The required function kernels for the network-decomposed KSOM are similar to those described in Section 6.4, listed in Table 6.13. For dataset decomposition, the required weight changes are computed and stored in a temporary array, costing time t_{dw}. In Algorithm 7.5, the only difference is that weight changes are directly adapted. So `ChangeWeights` costs $t_{cw} = t_{dw}$. The function kernels are therefore:

kernel	NSC	GCEL	PX
t_{nei}	24.95	20.77	3.37
t_{fnd}	8.13	6.77	.474
t_{cw}	10.01	8.34	.520

Table 7.9: *Times for Kohonen function kernels (μseconds).*

and the expected calculation time for the KSOM per pattern is based on Equations (6.5) and (6.6):

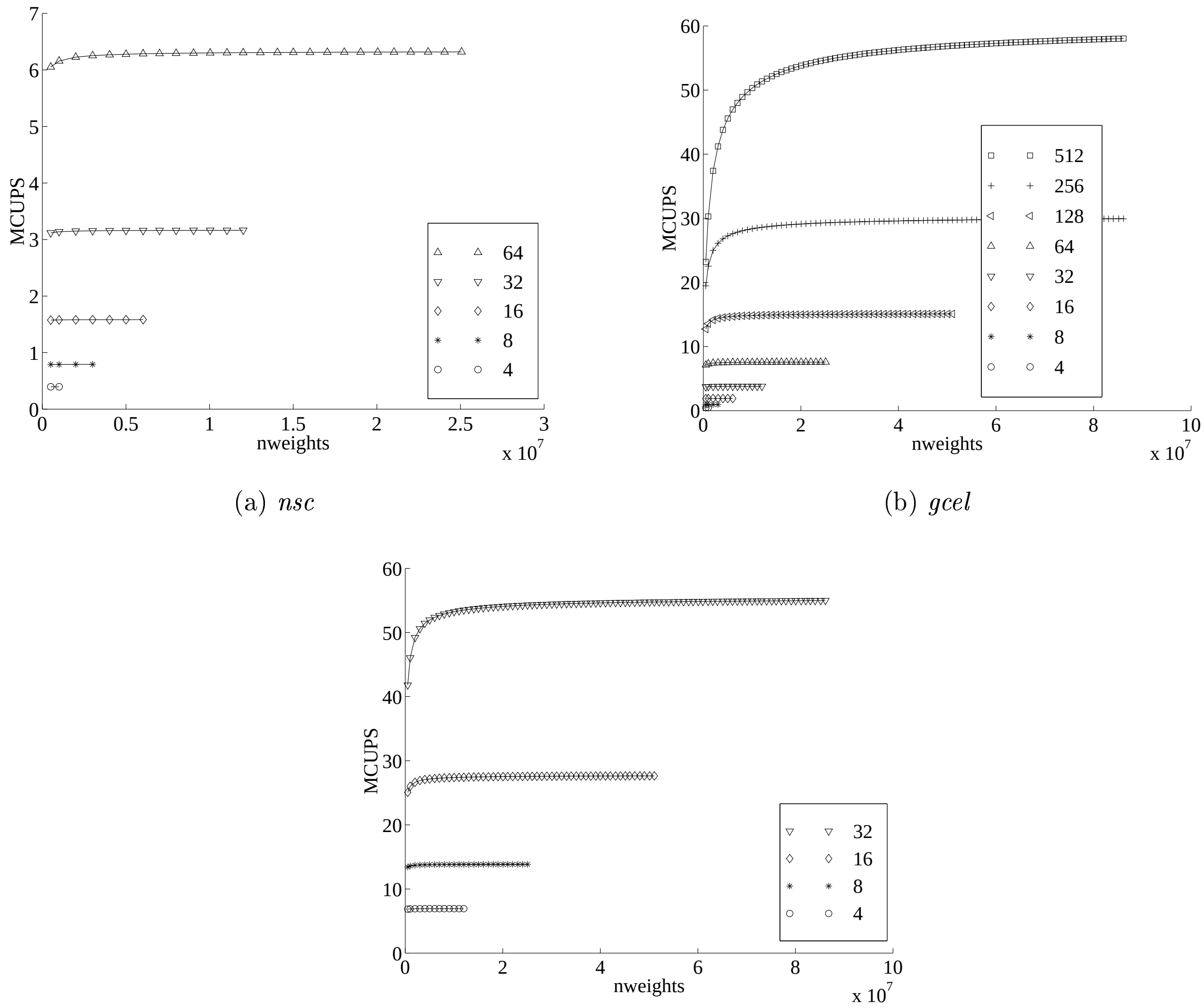
$$T_{calc}(P,n,w) \quad = \quad \frac{1}{P} \cdot (n \cdot t_{nei} + w \cdot (t_{fnd} + \pi/16 \cdot t_{cw})) \tag{7.15}$$

7.10 Performance and speedup

As can be expected, the performance of transputer networks is high for Kohonen neural networks. The only communications are gathering of the winning neuron and broadcasting new patterns. The latter communications can be ignored if the number of patterns is small enough to store all patterns on each processor. The communication time per pattern is:

$$T_{comm}(P,n,w) \;=\; T^{GAB}(P,2)+T^{B}(N) \tag{7.16}$$

For a wide range of synthetic KSOM neural network sizes, measurements were taken on the NSC, GCEL and PX. These were all accurate within less than 10%, which is due to the small amount of communication overheads. Below, the expected performance of diverse KSOMs are depicted.



(a) *nsc* (b) *gcel*

(c) *power*

Figure 7.12: *Expected performance for NSC, GCEL and PX.*

Note that these plots have similar characteristics as the performance plots depicted in Chapter 6; a certain performance limit is reached at the point where the problem size is big enough to neglect communication.

Table 7.10 below contains the speedup for NSC, GCEL and PX grids.

Mw	**NSC**					**GCEL**				**PX**			
	P=4	P=8	P=16	P=32	P=64	P=64	P=128	P=256	P=512	P=4	P=8	P=16	P=32
.5	4.00	7.99	15.92	31.53	61.25	60.17	107.77	165.59	198.88	3.97	7.77	14.57	24.54
1	4.00	7.99	15.95	31.70	62.25	61.55	114.54	190.80	257.88	3.98	7.85	15.06	26.82
2	4.00	8.00	15.97	31.81	62.90	62.44	119.25	210.96	316.90	3.99	7.91	15.40	28.52
3	4.00	8.00	15.98	31.86	63.16	62.81	121.22	220.16	348.45	3.99	7.93	15.53	29.26
4	4.00	8.00	15.98	31.88	63.31	63.02	122.40	225.91	369.93	3.99	7.94	15.61	29.72
5	4.00	8.00	15.98	31.90	63.40	63.15	123.13	229.61	384.55	3.99	7.95	15.66	30.01
6	4.00	8.00	15.99	31.91	63.47	63.25	123.69	232.45	396.24	3.99	7.95	15.70	30.23
7	4.00	8.00	15.99	31.92	63.52	63.32	124.08	234.46	404.76	3.99	7.96	15.73	30.38
8	4.00	8.00	15.99	31.93	63.57	63.38	124.42	236.22	412.42	3.99	7.96	15.75	30.52

Table 7.10: *Speedups*

Near linear speedups are achieved for the KSOM. Only for smaller neural network sizes or for the PX, lower speedups are estimated. This is because the computation/computation ratio becomes less in these situations. Scalability and efficiency are not considered here, because it is obvious that these are high when observing these speedup rates.

7.11 Network decomposed Satdat

The neural network architecture for Satdat is a 6x50x50 KSOM network. The number of connections required for this application is 15000, which is much smaller than the networks used for the experiments described above. However, the performance and speedup estimated based on Equations (7.15) and (7.16) are still relatively good for smaller processor networks:

	4	8	16	32	64	128	256	512
performance NSC	0.28	0.56	1.10	2.12	3.82	5.91	7.16	6.71
speedup NSC	3.99	7.95	15.70	30.23	54.42	84.28	102.05	95.69
performance GCEL	0.34	0.67	1.32	2.50	4.38	6.42	7.24	6.43
speedup GCEL	3.99	7.94	15.61	29.72	51.99	76.17	85.93	76.34
performance PX	3.45	6.60	11.56	16.81	18.61	16.17	12.27	8.83
speedup PX	3.93	7.50	13.15	19.13	21.18	18.39	13.96	10.04

Table 7.11: *Performance and speedup for Satdat*

For larger processor networks, the speedup limit is reached. The number of processors at which the speedup limit is reached can be found similar to Equation (7.14) as:

$$\begin{aligned} a &= n \cdot T_{nei} + w \cdot (t_{fnd} + \pi/16 \cdot t_{cw}) \\ b &= 2 * \cdot (2 \cdot t_{comm} + t_{acc}) \cdot 2 + 2 \cdot t_{comm} \cdot N \\ P &= \left(\frac{2 \cdot a}{b} \right)^{\frac{2}{3}} \end{aligned} \tag{7.17}$$

giving the following results:

	P	$\mathcal{S}(P)$	$\mathcal{P}(P)$	$\mathcal{E}(P)$
NSC	295	103	7.2	0.35
GCEL	246	86	7.3	0.35
PX	58	21	21.2	0.36

Table 7.12: *Maximum speedup and corresponding performance and efficiency for Satdat.*

7.12 Conclusions

The implementation aspects and measured performance results of two distinct neural network algorithms are discussed in this chapter. It is pointed out that in general, network decomposition introduces communication overheads. Therefore, care has to be taken to come up with an implementation that reduces communications. For backpropagation, two techniques are introduced for this goal: 1) a new distributed gathering technique using pipelining, and 2) an implementation of the backward pass which requires communications in the order of $\mathcal{O}(n)$.

For the KSOM, the communication overhead is relatively small, resulting in near linear speedup and high performances for large processor networks. For both the backpropagation and the KSOM it is observed that for smaller processor networks and smaller neural networks, the performance drops and the efficiency is low. Because in general, real applications are of moderate size, it must be concluded that no large processor networks must be used for network decomposition for such problems. In particular because modern transputer systems (PX) have a low computation/communication ratio.

For the two real applications Satdat and Nettalk, it appears that the maximum performance that can be reached is well within the available number of processors of each of the three target platforms. Using Equations (7.17) and (7.14), this performance can be computed, resulting in efficiencies between 0.3 and 0.5.

8

Neurosimulators

Outline

In the introduction of this thesis, a classification of users of neurosimulators is given. Furthermore, the neurocomputing environment is introduced and the activities with which users are occupied when doing neurocomputing are identified. A neurosimulator is a toolbox containing tools for developing, monitoring, manipulating, controlling and executing neural network simulation programs. In this chapter an overview over the features most neurosimulators have in common for providing these tools is given. The way in which neurosimulators access data from a running neural network simulation program is explained and it is argued that implementations based on features such as a network description language and hierarchical neural network data structures are less efficient than what can be achieved when using dedicated, tailor-made implementations.

8.1 The neurocomputing environment

In order to arrive at a general classification of neurosimulators and to examine the support that they offer to perform neurocomputing, in this chapter the specific features are listed that are encountered in environments that use neural networks. Figure 8.1 depicts such a neurosimulator environment.

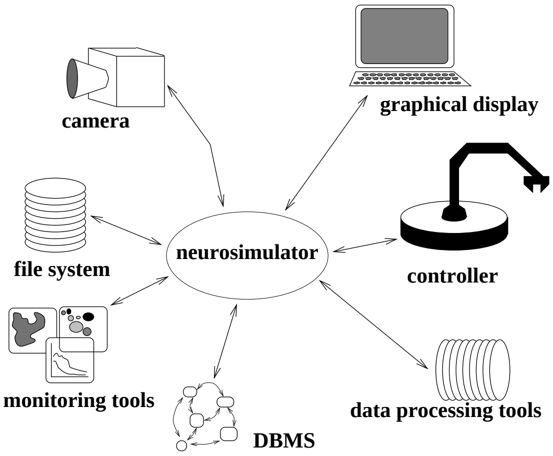


Figure 8.1: *A general neurosimulator environment.*

The neurosimulator can receive input from devices like a CCD camera with frame grabber, database management systems, a data generation program, or from file. It can send its outputs to data post-processing tools, output devices like a robot controller or graphical display, or to file. Via a user-interface the neurosimulator can be controlled. The neurosimulator can be either *explicit*, i.e., running as one program coupled to its surrounding environment, or it can be *implicit*, i.e., running as part of a larger program. Some explicit neurosimulators offer possibilities to dump the code of a tuned neural network as a stand-alone program which can be used to integrate it as an implicit program part.

8.1.1 Environments: user perspective

The user groups involved in neurocomputing and the neurocomputing life cycle as identified in the introduction of this thesis, are listed in Table 8.1:

user groups	**phases in life cycle**
model builders	initiation
tool builders	tuning
applied researchers	testing
end users	operation

Table 8.1: *User groups and life cycle of neurocomputing.*

A neural network is often considered as a black box that receives inputs, adapts itself based on features in the inputs, and produces outputs. In a operational phase, the black box model is acceptable for users, in particular for end users and applied researchers. For a neural network in operation, the user is not interested in what happens inside the black box, and the main efforts that are performed by him are inputting and outputting data into and from the black box. Important features that the neural network simulation exhibits in such a setup are that it is integrated with data preprocessing, post-processing and/or data evaluation tools. However, for model builders it may be extremely interesting what goes on inside the box, as one of their goals is to understand the functioning of the brain. Also for applied researchers in the tuning and testing phases — during which the adaptation process of the neural network takes place — the black box concept is not sufficient. Though there exist neural networks which are more or less self-organizing, like ART [12], fieldnet [85] or Kohonen [58], in order to monitor and control the adaptation process it is required to be able to halt a running session and edit or view the network parameters.

When considering these aspects and the different user groups for neurosimulators, three environments can be distinguished:

1. *Production environments.* Such an environment in most cases contains implicit neurosimulators. It is integrated in a larger workbench and carries out one step in a range of processing stages. Note that before the end application is implemented, the complete neurocomputing life cycle could have been run through. This can be performed by some hired experts or by employees from the research department of the company. In either case, if neurosimulators are used in production environments, they must somehow have a means to be integrated in the environment. This means that it must be possible to input and output data in certain formats, or to extract stand-alone neural network code which can be used in the end application.

2. *Model research environments.* Model builders impose more requirements on neurosimulators. They have a lot of experience in building, tuning, evaluating and applying neural networks, and want a neurosimulator to support them with these tasks. Whereas in production environments dynamic I/O is one of the salient features, for this environment inputs and outputs are obtained from relevant training and testing datasets, or from artificial data. Knowledge from the data (through data processing and statistical evaluation) can be built into the models or can be used in some preprocessing stage.

3. *Application research environments.* These environments comprise the former two, as the eventual goal is to exploit neural networks for a specific application, and before this is achieved, a lot of experimentation has to be done.

Whereas in the first environment in general not much experience will exist with using neural networks, in the other two environments this often is the case. It may even be so, that researchers already have a number of existing implementations of a set of neural network algorithms in their department. In order for them to accept a new neurosimulator, they will require that it is possible to incorporate their existing code within the new system, without too much effort.

8.1.2 Environments: neurosimulator perspective

In [80], a taxonomy for neurosimulators is given by Recce, Rocha and Treleaven. Three classes of neurosimulators are distinguished based on the support that they offer: application-oriented, algorithm-oriented and general programming systems. The characteristics of each of the classes are determined by the intended user and scope of the application.

Application-oriented systems are targeted on end products. They are used in production environments, and often have integrated the neurosimulator as an implicit version into the application. The end product is developed based on requirements specified by the end user. The neural network part of the application is either implemented from scratch, by an *application-specific* neurosimulator, or by a *general-purpose* neurosimulator. The first type is developed specifically for the end user or for the application area he is interested in. This means for example that it is equipped with means for loading and storing data according to the formats used in the target environment. The second type are commercially available packages, generally targeted on PCs, like NeuralWorks, Nestor, and BrainMaker. Such neurosimulators provide an easy to use user-interface via which a range of neural network models can be controlled.

Algorithm-oriented systems are neurosimulators dedicated for one specific neural network model or neurosimulators containing algorithm libraries. Many algorithm-oriented neurosimulators are targeted on multi-layered perceptrons[1]. Others are targeted on for example Kohonen networks [60, 59]. Because these systems are tailor-made, in general they contain highly efficient implementations of the model or application that is supported. Algorithm-oriented systems are used if a user knows or expects that his application can be solved by a particular neural network. The disadvantages are that they are not very flexible; some difficulties may arise if they have to be integrated in the applications environment, or if the user wants to make changes to the neural network algorithm.

General programming systems have the advantage over application-oriented and algorithm-oriented systems that they offer possibilities for researchers to implement new models. They can be divided in *development programming systems*, *open systems* and *hardware-oriented systems*. Neurosimulators belonging to the first category provide means to construct neural networks using sets of library routines or using a neural network description language (NDL). Programs using the library routines have to be linked with the neurosimulator, programs specified following the NDL have to be interpreted or compiled and run. Both methods have the advantage that the neurosimulator may safely assume that the neural network model is specified in a prescribed manner. This means that it can access the

[1] Even many general-purpose neurosimulators, though offering features to incorporate other models, have only implemented the backpropagation network.

datastructures, provide monitoring and manipulation tools and that it can use its own model of execution and control. The second category, open programming systems, differ from the previous one by offering the facility to be customized, for the benefit of the user or for the application. Changing the environment can result in a customized user-interface and/or integration with user-defined tools. The third category are hardware-oriented systems, which are able to map a given neural network specification on a given hardware configuration. A comparison between three neurosimulators belonging to the third class is given by Plonski in [77]. The Rochester Connectionist Simulator, Genesis and Sfinx are compared. Plonski also distinguishes three classes of general programming systems. The first class contains demonstration programs which are dedicated to one particular neural network paradigm to (briefly) demonstrate its behavior. The second class are special purpose neurosimulators, intended for a broader range of experimentation for a set of network models. The third class are simulation environments, which differ from the previous two by supporting a number of network models and by providing facilities to construct new models. In his paper, Plonski discusses the typical features neurosimulators exhibit. An overview over these features is given in the next section.

8.2 Features of neurosimulators

As stated before, a neurosimulator must be able to aid the user with the construction, tuning, testing, integration in an environment, and execution of a neural network. Depending on which class of neurosimulator is used, some of these tasks may be more or less supported. For example in [77], [80], [109] and [116] the characteristic features that neurosimulators offer are listed. Neurosimulators can have a graphical user-interface, an algorithm library, support for building new models, application specific tools, dedicated hardware accelerators, etc. In figure 8.2, a general purpose neurosimulator is depicted. It consists of tools to construct, visualize and manipulate, control and execute a neural network executable. The executable may be connected to different tools through a data interface, or it can be controlled by a (graphical) user-interface. The following components can be identified:

- ◇ *User-interface.* Via a graphical user-interface, or sometimes a character based command-line interface, the neurosimulator can be controlled. The user interface supports starting up actions such as learning or recalling a set of patterns. Furthermore, it supports the visualization of neural network specific data, by starting up graphical monitor or plot tools. In many occasions, user commands issued via the user-interface can be logged to a file which can be loaded to perform some kind of batch processing.

- ◇ *Algorithm library.* Neurosimulators often contain a parameterizable algorithm library, consisting of various implementations of neural network models. These can be called from the user-interface, but can also be used to develop application specific user programs. Typical parameters for initiating the algorithms are architectural settings determining the structure of the neural network, control parameters such as the number of iterations

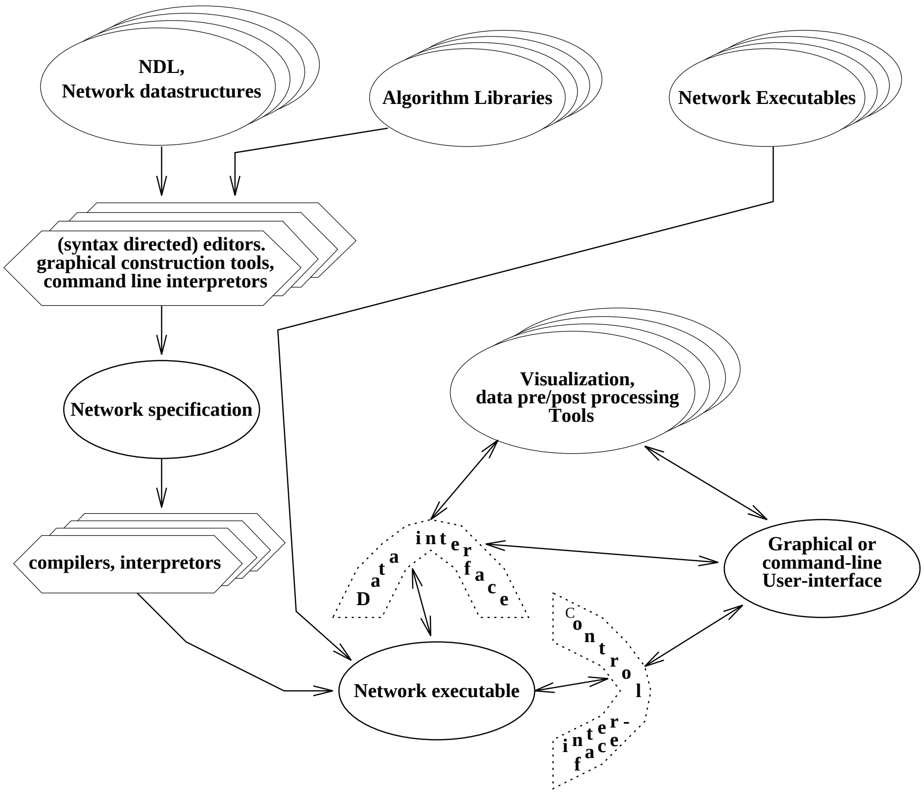


Figure 8.2: *A general purpose neurosimulator.*

to use for training, and learning parameters like the learning rate, determining the speed of learning. Integrating algorithms from the library as stand-alone programs can be supported by neurosimulators. In most cases this is done by dumping the code of a tuned neural network simulation program with static values assigned to the weights, thresholds and other parameters into a stand-alone program.

◇ *Support for building new neural network models.* New neural network models can be specified via library routines using a hierarchical neural network datastructure, or by using a neural network description language. As each different implementation uses the same kind of datastructures, the graphical user-interface is able to connect to objects hidden inside the network. In this way, objects can be changed or monitored. Some neurosimulators support graphical construction of neural networks by drawing icons representing neurons or clusters, and arrows for connections between them.

◇ *Network monitoring tools.* Graphical monitoring tools initiated by the user-interface allow the visualization of all kinds of information from the neural network. This could be the weight values, neuron activations, parameter values like learning rates, etc. Often, graphical plot tools are provided to monitor the performance of the network error during training. A lot of neurosimulators offer the possibility to dump neural network data in a specified format which can be used by the user to transform it to the formats required by his specific tools. Others allow the possibility to dump data into standard formats like PostScript[2], Matlab [66] or gnuplot [57].

◇ *Dedicated target platforms.* In order to cope with the processing and memory requirements required by todays neural network applications, a wide range of high performance target platforms are proposed and being used. Some neuro-simulators use parallel processor systems (Tollenaere [100], Richards [82], Recce [80] , Goddard [40]), some are targeted on super computers (Ghosh and Wang [38]), others on dedicated neuro asics (Theeten [99], Han [43]) or use heterogeneous architectures (Duranton [27]).

◇ *Application specific tools and devices.* For most experiments using neural networks, the information produced by the tools mentioned above is adequate for making general statements about how the network is performing during training. However, in order to find out how well a network actually behaves after training, it will have to be used for production. Often, this involves coupling the network simulator to some I/O device or controller, or feeding it with large amounts of data. As the data can have any representation required by the application at hand, this involves formatting it to the formats used by the neurosimulator. Image processing applications for example, require the simulator to be coupled to graphical display programs. For information systems, it is coupled to database management systems. For real-time applications, it has to be coupled to the corresponding devices. Figure 8.1 in Section 8.1 depicts a number of possible tools and devices to which a neurosimulator can be connected. Most neurosimulators have no support for dynamic I/O, i.e., they have no means to receive inputs from preprocessing tools nor to send outputs to post-processing tools online. Integration in such cases means that an explicit neurosimulator has to be transformed in an implicit one by extracting stand-alone neural network code.

8.3 The traditional neurosimulator engine

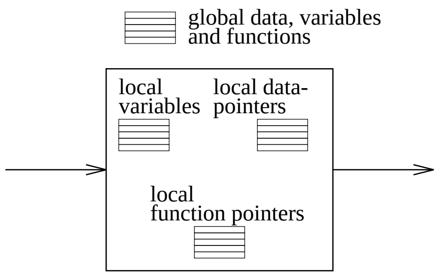
Finding out how neural networks can be constructed and run, how data processing and visualization tools can access neural network specific data and how this data can be manipulated, boils down to finding out about the internal datastructures and flow of control of the neural network simulation program. In the traditional approach, there are three ways via which neural network objects contained in a program are accessed. The first is completely hidden from the neural network experimenter. Via an NDL or a script language, he is able to add, delete or manipulate neural network objects. To enable this, the simulation kernel is built using a hierarchical datastructure, e.g. comprising structures or classes like *network*, *neuron*, *connections* and *activation functions*. The second is a more open approach where the user may program his own applications, using library routines to manipulate the neural network datastructures. These routines (or macro definitions)

[2]PostScript is a trademark of Adobe Systems, Inc.

provide an encapsulation of the general neural network datastructures for the user. The last is a completely open approach, where a neural network programmer has full access to the datastructures.

8.3.1 The general neural network datastructure

In the black-box model, a neural network transforms inputs into outputs, possibly while adapting itself based on the inputs. Black boxes can be nested, i.e. a black box may contain several other black boxes, each representing a specific part of the neural network. For example, a three-layer backpropagation network could be described by a hierarchical structure of network, three layers, several neurons and connections. The black box *network* contains three black boxes, each representing a *layer*. Each layer may contain an arbitrary number of *neuron* black boxes and connections may be defined between layers and individual neurons. Note that using this hierarchical approach, the concept of connecting for example layers may be implemented by fully connecting the outputs of the first layer to the inputs of the second layer. In a simulation of such a conceptual neural network model, each black box definition can be a structure (or class), containing local and global function pointers to routines which implement its behavior, and global and local data pointers and variables for storing and retrieving information.

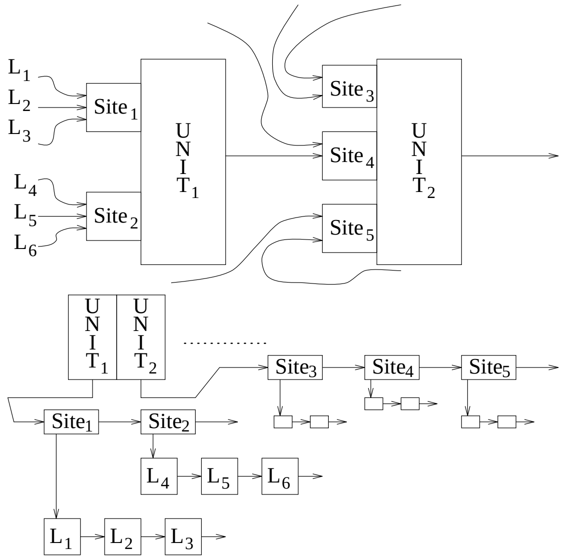



```
typedef struct {
   type_1 variable_1;
       ...
   type_m variable_m;
   type_1 data_1;
       ...
   type_n data_n;
   type_1 function_1;
       ...
   type_o function_o;
} BlackBox;
```

Figure 8.3: *The general black box concept.*

For example, consider the neuron model of the Rochester Connectionist Simulator [40]. Each neuron is called *unit*. Units can have a number of sites, each *site* handling a number of incoming *links*. An excerpt of the unit model, its datastructures and structure specifications (from [40]) are depicted below[3].

[3]The fields in the datastructure do not precisely reflect the names specified in [40]




```
typedef struct {
   Potential;
   State;
   Output;
   *Data;
   *Neuron_Function;
   *Site_List;
        ...
} Unit;
```

Figure 8.4: *The RCS unit model and unit datastructure.*

Units are accessed via an index in an array of units. A neuron's input is determined based on the values of its sites. Each unit has a linked list of sites. Incoming connections (links) are implemented as a linked list of links, which are stored by each individual site. The RCS provides a set of (command interface) routines via which a programmer can build a neural network via the command line interface, like:

```
MakeUnit type function,
AddSite index type function,
MakeLink index_from index_to value data function
```

However, this is a time consuming process and it is advised to build neural networks in a C-program. For this purpose, an extensive library of routines that allocate, build and connect neural network objects is provided. Routines with similar syntax like the ones above exist. Furthermore, a number of syntactic convenience routines can be used to access the neural network. Given a unit with index `idx`, for example its output, state or data fields can be requested or updated via:

```
GetOutput(idx);         GetState(idx);         GetData(idx);
SetOutput(idx,value);   SetState(idx,value);   SetData(idx,value);
```

A new neuron, site or link can be constructed by implementing programmer-defined functions. Each of these 'black boxes' has a field `Data`, which in fact is a `void` pointer and can be used to contain any information. This involves that if newly defined functions use this data, during allocation of a black box (e.g. via `MakeUnit`) also the data for the unit has to be allocated.

The EENS [62] neurosimulator developed at our department has a similar conceptual model like the RCS. Its neuron black box is depicted in figure 8.5.

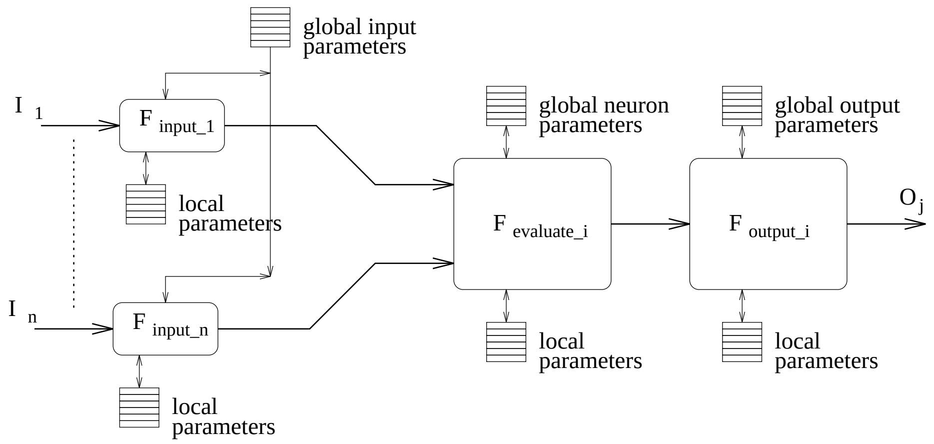


Figure 8.5: *The EENS neuron model.*

Also the Pygmalion [2] neurosimulator and many others like Aspirin/MIGRAINES [63] and Genesis [122] are built based on a general hierarchical neural network datastructure. Pygmalion datastructures contain objects like system, network, layer, cluster, neuron and connections. Macros are provided for accessing data, like `n_0_2_3_6.0` , which produces a neuron's state value:

```
sys->net[0]->layer[2]->cluster[3]->neuron[6]->state[0]
```

Genesis has a biologically very plausible model of neural networks. Consider for example how some of the fields of a unit in Genesis are defined:

```
typedef struct {
    char               *name;
    int                 index;
    struct object_type *object;
    Element            *parent;
    Element            *child;
    Element            *next;
    double              state;
    float               Rm;       /* membrane resistance */
    float               Cm;       /* membrane capacitance */
    float               Em;       /* reversal potential */
    float               inject;   /* injected current */
} unit_type;
```

In Aspirin/Migraines a neural network is constructed based on what actually is called a black box. A black box can be defined by a `DefineBlackBox` statement, which requires a description of its constituents:

```
DefineBlackBox <name> {
   OutputLayer    /* where to put output data */
   InputFilter    /* C function producing net input */
   OutputFilter   /* C function producing output */
   Components     /* list of layers containing neurons with connections */
}
```

8.3.2 Access and control of neural network data

Because each neurosimulator is built based on this concept of hierarchical datastructures, accessing a value of a neural network boils down to following the pointers in the structures. The main advantage of the black box models described above, is that they are general purpose. Pointers to attach any user-defined data or functions are available. Furthermore, as for each neural network the same datastructures are used, the neurosimulator is able to access them and for example start up visualization tools for monitoring (parts of) the network. The disadvantages are that programmers have to stick to the syntax of the datastructures and associated macro definitions and library routines, and furthermore, that this model uses a rather inefficient flow of control. For example, consider algorithm 8.1. It implements a simplified neuron black box model and training algorithm for the backpropagation network. Each neuron object has a number of properties, defined in a C-structure. It contains variables maintaining its status, like delta and activation. It contains parameters denoting the number of inputs and outputs and indexes to the first neuron from which it receives inputs and to which it sends outputs. Finally, pointers exist to its incoming connections and to three functions which implement the computation of a neurons input and activation values and the training mechanism.

```
typedef struct {
   float netinp;
   float bias;
   float activation;
   float delta;
   int ninputs;
   int first_input;
   int noutputs;
   int first_output;
   float *connections;
   void (*input_function)();
   void (*output_function)();
   void (*train_function)();
} Neuron;
```

(a) *The neuron object properties.*

```
Pattern *pat;
Neuron *neurons;
int npatterns, nneurons;
void epoch() {
   int i,p;
   for (p=0;p<npatterns;p++) {
      for (i=0;i<nneurons;i++) { /* forward pass */
         neurons[i]→input_function(&neurons[i],pat[p]);
         neurons[i]→output_function(&neurons[i],pat[p]);
      }
      for (i=nneurons-1;i≥0;i--) /* backward pass */
         neurons[i]→train_function(&neurons[i],pat[p]);
   }
}
```

(b) *The basic training algorithm.*

Algorithm 8.1: *Backpropagation using hierarchical datastructure.*

In this black box model, the flow of control is ruled by the routine `epoch()`, which knows about the number of neurons and which functions to call. Running this algorithm in most cases is less efficient than running a dedicated implementation of the backpropagation neural network. The first reason for this is because of the flow of control. For each neuron a number of function calls are required. Especially for a large number of neurons this introduces overhead compared to running dedicated in-line code. Note that this overhead increases for more complex models like the RCS or EENS model described above. The second reason is the way in which data is accessed. The more references in a hierarchical datastructure are required to access e.g. an activation or weight value, the more time is required.

8.3.3 The neural network description language

Neurosimulators like Pygmalion, Genesis and Aspirin/Migraines provide an NDL for specifying neural networks. The first provides the high level language nC and an intermediate level language nC-code. The second has defined the language Aspirin, which can be used in close combination with the MIGRAINES interface. Genesis provides shell-like commands via which neural network objects can be defined, connected and executed. The commands can be stored in a shell script which can be run in batch mode.

A network description language offers flexibility combined with encapsulation of programming efforts. Using predefined modules or customized library routines (e.g. programmed in C) contained in an algorithm library, with the NDL new neural network models can be built. A NDL script can either be interpreted (Genesis) or be compiled into executable code (Pygmalion, Aspirin/Migraines). The former method results in a slow execution. The latter offers the advantage of having several compilers building code for diverse target platforms. For example the goal of Pygmalion was to have compilers that — based on a NDL and a description of the target machine configuration — produce code for transputer based multi-processor systems.

The underlying datastructures are the same as discussed in section 8.3.1. This involves that using an NDL is not as efficient as when using dedicated code. Especially if the goal is to automatically decompose a neural network program over a parallel processor architecture this is the case. This problem can be translated to the problem of mapping two connected graphs onto each other, for which no general optimal scheme exists.

8.3.4 The graphical user-interface

As mentioned before, via the user-interface the user can control a running neural network simulation program. This comprises initiating, halting and continuing actions like loading patterns or a training and recall. This also includes monitoring and changing parameters associated with these actions.

An important observation made when considering existing neurosimulators and their environments was made in [116]. The amount of code actually needed to implement a neural network's dynamics is far less than the amount needed for implementing the user-interface mechanism for controlling the network. In some cases the required code for the user-interface may take up more than 90% of the total code. Especially, this holds if the neurosimulator also provides graphical monitoring and debugging tools.

8.3.5 I/O and neurosimulators

A final common feature of neurosimulators discussed here are I/O. A running neural network simulation can input data or generate output to be loaded or output from file or some dynamic I/O channels. For example consider Figure 8.1, which depicts such a set up. Most neurosimulators have defined one or more *pattern formats* for loading and storing data to be processed. In general, such a format contains a header describing the number of patterns, the number of input features per pattern and — if supervised data is concerned — the number of output classes. The data is generally organized in input/target pairs. For example, the patterns for the XOR problem would be specified as:

```
4 2 1
0.1  0.1  0.1
0.1  0.9  0.9
0.9  0.1  0.9
0.9  0.9  0.1
```

For certain applications, specific requirements may exist which cannot be solved via such pattern files. In particular applications in sensor/motor fusion and vision require for the neural network to input or output data to or from hardware devices. Current neurosimulators offer the possibility to add hooks to so called input functions or input routines, as explained in the previous sections. Programmers can add their application-specific I/O routines, written in, e.g., C to implement the I/O. For each execution of an input or output routine, the execution mechanism of the neurosimulator checks if some programmer-defined routine has to be called. Upon calling such a routine, the neural network is supplied with the corresponding input data or outputs the corresponding output data.

Similar considerations can be made toward routines for control and visualization of a neural network simulation. General purpose neurosimulators have utilities to view the contents of neural network datastructures such as the activation of units and the weight values. If the visualization of data requires a different layout, specialized visualization routines can be attached to the corresponding data structures.

8.4 Conclusions

Considering the observations discussed in the previous sections, it can be concluded that:

1. General purpose neurosimulators are built on the concept of a hierarchical data structure.

2. Using this structure, a user/programmer can exploit algorithm libraries, a NDL, or dedicated (C)-code to add or modify part of a neural network simulation program.

3. This means that a user/programmer has to adhere to a predefined syntax and data structures, which in general is not as efficient as customized code.

4. Application specific I/O interfaces and neural network simulation code may also suffer from this situation.

5. On the other hand, general purpose tools may be designed based on these data structures.

6. Research and applications of neural network simulations require an intensive access to the simulation code. Changing the model or adding I/O-specific routines is an activity with which users of neurosimulators are heavily occupied.

7. It is also noted that the control over a neural network simulation is relatively constant. A limited set of actions such as loading of a network and data, and starting up training sessions are to be controlled via the user-interface.

8. Furthermore, the vast majority of the code required for neurosimulators is occupied by the code required for the user-interface.

9. And finally, neurosimulators require a means to be coupled to the outside world; to a user-interface for controlling the simulation, to input data provided by input tools, and to output data needed by output tools.

9

An action-oriented neurosimulator

Outline

In this chapter, the design and implementation of a so-called action-oriented neurosimulator is discussed. The notion of actions and associated objects and attributes is presented and based on this, the idea of program descriptions is introduced. Using these concepts, a set of interface definitions is specified via which the manager of the neurosimulator environment (called CONVIS) can access data and can rule the flow of control of a running neural network simulation program. The neurosimulator PREENS consists of this manager CONVIS and a set of implemented tools and neural network algorithms. PREENS is introduced here and, as an example, its application to remotely sensed satellite imagery is described.

9.1 Objects and attributes of actions

It was argued in Chapter 8 that there exist only a limited number of *actions* that are carried out when using a neurosimulator for simulating artificial neural networks. The actions are concerned with loading data, with training, recall, and operation. Actions contain *components*, which can contain *objects* and *attributes* [116]. These can be further divided in parameters, options and settings for installing the initial configuration of an action, and variables and data whose values can be changing during execution of an action.

- ◇ *Objects* associated with an action represent *parameters*, *variables* and *data* that exist in the implementation of an artificial neural network. In most cases they represent real objects of the neural network, such as activations, thresholds and weights. Also, they may represent data structures required for implementing the neural network, like input/target patterns and derived components like a confusion matrix[1], or like a classification result. In other cases, objects may be more related to the program simulating the neural network. Examples of the latter case are parameters like the name of a file in which training patterns or weights and activations of a neural network are stored, or program variables like the current pattern or iteration number.

- ◇ *Attributes* are *settings* and *options*. They are required for specifying the flow of control of a neural network simulation program. Settings can be considered as radio buttons or boolean values. When a setting is `on` or `off`, this indicates that a part of the program has to be executed or not. Options are program variables representing one out of several alternatives. They can be considered as switch buttons switching between a number of states of which only one may be active.

Consider for example a training session of the multi-layered perceptron with momentum learning rule and sigmoid activation function [83]. Such a session would be implemented in an action `learn`. *Parameters* that rule the computation of weight error derivatives are the learning rate η and the momentum α. Other parameters can be the number of iterations or the error criterion after which training has to stop. The error generated by the neural network at a certain time can be expressed as the sum of squared differences between target and output activations. This error is a scalar *variable* that can change over time. Non-scalar values that can change over time are called *data*. Data contained in the neural network are for example activation and threshold values of the neurons, and the values of weights. During training, certain modes can be on or off. For example, it can be decided whether during training the neural network components have to be saved or not. The attribute that represents such a decision is called a *setting*. When giving the setting `autosave` the value `on`, this indicates that saving the net during training is required. Finally, certain *options* can exist that have one out of several values. Activating a neuron can be performed via, e.g., a binary threshold function, the sigmoid function, or the `tanh` function. This can be specified by an option called `activation_function`.

[1] Confusion matrices will explained in more detail later in this chapter.

For the objects and attributes of the action `learn` described here, the following identities can be listed:

parameters	learning rate, momentum, nr of iterations, error criterion
variables	net error, current pattern, current epoch
data	activations, thresholds, weights
options	activation function
settings	epoch training, autosave

Table 9.1: *Sample objects and attributes for an action* `learn`.

As another example, consider an action `load_network`. A parameter for initiating this action can be the name of a file from which to load a network's weights, thresholds or activation values. If no previously stored neural network has to be loaded, but a new network has to be initialized, parameters can be the minimal and maximum values between which weights have to be initialized. The option indicating whether a new or an existing network has to be loaded could be called `new/from file`. Parameters determining the architecture of the network are the number of layers, and for each layer, its number of neurons. Data are the weights, activations and thresholds. Note that for this action, no variables or settings are required.

Throughout this chapter, further definitions and examples of actions will be given. In some occasions, these will be accompanied by explanations how they can be controlled via the graphical user-interface of CONVIS. The CONVIS main window is depicted in Figure 9.1. Actions are divided in the categories I/O (e.g., `load_network`, `save_network`, `load_patterns`), `learn`, `recall` and `classify`. The latter actions correspond with the tuning, testing and operation phases in the neurocomputing life cycle. For actions that do not fit in this concept, the `miscellaneous` category is provided.

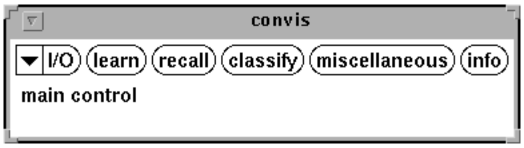


Figure 9.1: CONVIS *main window. It contains a pull-down menu called* `I/O`, *and it contains buttons via which control for an action can be activated. Upon activating an action, the so-called action control window pops up (see Figure 9.2).*

9.2 Actions and program descriptions

The observations made in the previous section hold for all neural network simulation programs. Each action implemented for a program can be described via parameters, variables, data, settings, and options. And by describing each program's actions, the complete program can be described. Different neural network models may have different objects and attributes for an action. The momentum parameter for backpropagation does not exist in the Kohonen neural network. Vice versa, the range of the neighborhood function is a parameter that is not used in the backpropagation neural network. So whereas different neural network programs implement the same actions, it is the specification of their actions that may differ.

A program description describes the actions implemented for a certain neural network simulation program. Each action is described by specifying its components; its objects and attributes. The syntax of a program description is depicted below:

```
program_description:  actions
actions:              empty | action; actions
action:               'action' name '{' runmode components '}'
components:           empty | component; components
component:            object | attribute
object:               parameter | variable | data
attribute:            option | setting
```

In PREENS, each neural network simulation program, say `sim.c`, has to be specified by a program description, say `sim.descr`. In section 9.3, it is described how a running simulation program from the PREENS algorithm library (say *sim*) and CONVIS interact. In an initialization phase, CONVIS interrogates *sim* for the actions it contains. The simulation program parses its program description and builds up a data structure containing the specification of its actions. When parsing a program description, for each action encountered, an entry in an array of `CAction` structures is allocated. Such an entry contains the following fields:

```
typedef struct {
    char name[C_NAMELENGTH];  /* name of the action, e.g., 'learn', 'recall' */
    int id;                   /* identifying the action, index in array */
    int runmode;              /* tells if action is interruptible of not */
    int nparameters;
    CParameter *parameters;
    int nvariables;
    CVariable *variables;
    int ndata;
    CData *data;
    int noptions;
    COption *options;
    int nsettings;
    CSettings settings;
} CAction;
```

Control over an action can be instantiated via the CONVIS main window by clicking on the action or selecting it from the pull-down menu (see Figure 9.1). The action control window is activated via which the components of an action can be selected, and the execution of an action can be controlled.

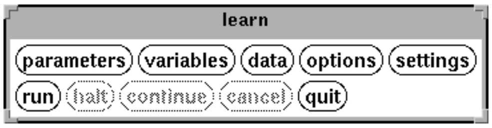


Figure 9.2: *Action control window for the action* `learn`. *It contains buttons via which control can be initiated for its components. And it may contain control buttons via which an action can be started, halted, continued, canceled and quit.*

An action is interruptible if it can be halted during execution of the action. For some actions it does not make sense to interrupt it, for example for an action `save_network` this is the case. In the program description of an action, with `runmode`, it can be specified if an action is interruptible or not. For actions that are not interruptible, only two control buttons are generated; the `run` and `quit` button. Note here, that CONVIS dynamically installs itself based on the program description. This is a feature that will be described in more detail later in this chapter.

In the next sections, the data structures of all components are defined, and it is explained how a component must be specified in a program description.

9.2.1 Parameters

In a program description, any number of parameters can be specified for an action. For each parameter, an entry in the array `parameters` is allocated for storing the parameter descriptions. The data structure for a parameter is given below:

```
typedef struct {
     char name[C_NAMELENGTH];      /* e.g.  'niterations', 'lrate' */
     int id;                       /* index in array 'parameters' */
     int type;                     /* e.g., int, float, double, string */
     double value;
     double min;
     double max;
     int editable;                 /* hint for CONVIS */
     char string[C_STRINGLENGTH];  /* if type is string, contains the string */
     int size;                     /* # of bytes the parameter occupies */
} CParameter;
```

When clicking the button `parameters` in the action control window, the parameter control window pops up:

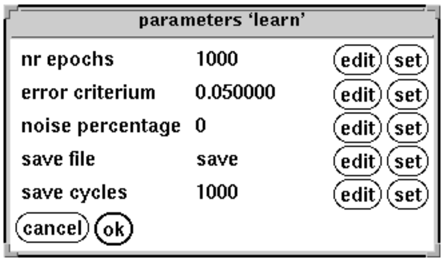


Figure 9.3: *Parameter control window of* CONVIS.

Via this window, values of parameters can be edited and viewed. When clicking the `edit` button, a window pops up via which more fields of the corresponding parameter can be viewed or edited (see Figure 9.4). For each field, it can be specified in the program description whether it can be edited (like the `min` or `max` fields), or just viewed (like the `type`, `name` or `identification` field. The syntax for specifying a parameter in a program description is given below.

```
parameter:     parameter '{' name type value min max editable string_value '}'
name:          string
type:          C-type | file | string
value:         C-type
min:           C-type
max:           C-type
editable:      boolean
string_value:  string
```

A `C-type` is any type in the C programming language, such as `short`, `float` and `double`. Furthermore, the types `bit` and `byte` are contained in a `C-type`. A `string` can be any sequence of characters between quotes. The difference between type `string` or `file` is that for the latter, CONVIS adds a mechanism to start up a file browser, as depicted in Figure 9.4. For other types, this is not the case:

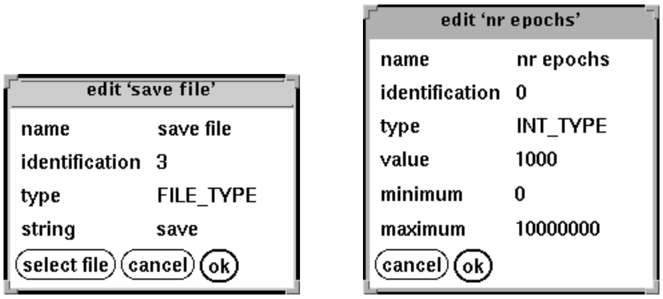


Figure 9.4: *Edit parameter windows of* CONVIS. *When clicking* `select file`, *a file browser pops up.*

9.2.2 Variables

Variables and data may change in time, and in their corresponding data structures, fields are contained for maintaining information associated with monitoring the changing values. In section 9.3, this is explained in detail. The items that have to be specified for a variable in a program description are similar to those for a parameter, but variables can have no type `string` or `file`:

```
variable:  variable '{' name type value min max editable time_id '}'
```

The `time_id` field is a string (which may be empty), for indicating the unit quantity at which the variable is changing, e.g., second or iteration number. The specification of, e.g., a variable called `error` following this syntax is:

```
variable { "error" double 0.0 0.0 MAXDOUBLE 1 "epoch" }
```

9.2.3 Data

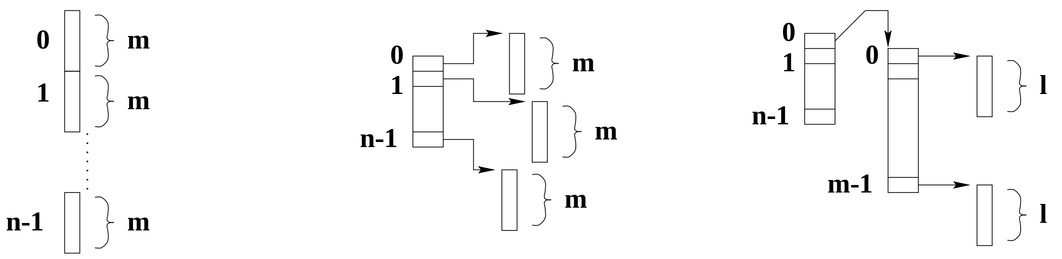
Variables can hold only one value, as they represent scalar objects. If more than one value is used within an object, it is considered as data. The `CData` structure that defines the data concept in PREENS is given below:

```
typedef struct {
     void *dataptr;                     /* pointer to place data is stored */
     CData_Description description;     /* describing the data */
     int allocated;                     /* is data allocated or not?  */
} CData;
```

Data can be structured any way a programmer wants to. However, it was argued that the more data is structured in, e.g., linked lists or hierarchical data structures, the more time is required for accessing its individual elements. Especially in neural network simulations, a high number of relatively simple operations have to be performed on vector or matrix elements. Based on these observations, the `CData` structure was developed. It contains a pointer to the place where the data is stored. Furthermore, it contains a description that specifies how the data is structured. The data description allows data to be structured in multi-dimensional arrays, thus supporting vectors and matrices in particular. Up to four-dimensional arrays of data vectors are allowed, covering the majority of data structures encountered in neural network simulation programs. Fields in the `CData_Description` structure that are used to describe the structure of data are:

```
typedef struct {
     int dimension;   /*  dimension of the data, either 1,2,3 or 4 */
     int depth[4];    /*  for each dimension, the nr of data elements */
     int length;      /*  the number of features per data element */
     int flat;        /*  indicates whether data is allocated consecutive, or
                          more-dimensional */
} CData_Description;
```

For example, consider a set of n input patterns, where each pattern has m elements. Two ways of arranging a data structure containing the nxm elements are depicted in Figures 9.5(a) and 9.5(b). A two-dimensional array containing element vectors of length l is depicted in Figure 9.5(c):



(a) *One dimensional flat nxm array.*

(b) *One dimensional non-flat nxm array.*

(c) *Two-dimensional nxm array with elements of length l.*

Figure 9.5: *Arrangement of three data structures.*

In Table 9.2, the way such data structures are described is given:

field	Figure 9.5(a)	Figure 9.5(b)	Figure 9.5(c)
dimension	1	1	2
depth[0]	n	n	n
depth[1]			m
length	m	m	l
flat	yes	no	no

Table 9.2: *Description of data depicted in Figure 9.5.*

In general, data is allocated dynamically. Before running an action, all the data it uses has to be allocated. As a program description is a static description of a program, in general the structural dimensions of data are not specified.

```
data:          data '{' name type struct_descr monitorable time_id '}'
struct_descr:  dim '['depth']' length flat
```

The data from Figure 9.5(c) could be specified as:

```
data { "input pattern" float 2 [m n] 1 0 "epoch" } or
data { "input pattern" float [] 0 "epoch" }
```

In the second specification, the description of its structural dimensions (the `dim`, `depth`, `length` and `flat` fields) is determined at runtime.

9.2.4 Options and settings

Options must be used if one out of a number of possibilities has to be selected. For example, if an action `learn` has implemented several learning methods, one of these can be chosen. Or if an action `load_patterns` allows loading of either training data, data for testing, or data to be classified, an option `which patterns` can be defined, which has state values {`0,1,2`}. These values could be represented at the user-interface via the strings `"load training patterns", "load test patterns", "load classify patterns"`. Such an option is used to indicate which out of three possible pattern sets has to be loaded. The definition of the `COption` data structure is:

```
option:  option '{' name nstates state state_names '}'
```

In this definition, `state_names` contains for each of the `nstates` states, the string representing a state. For the example of determining which set of patterns has to be loaded, the following is a valid specification:

```
option {  "which patterns" 3 0 ""load training patterns",
          "load test patterns", "load classify patterns" }
```

The option control window from CONVIS is depicted in Figure 9.6:

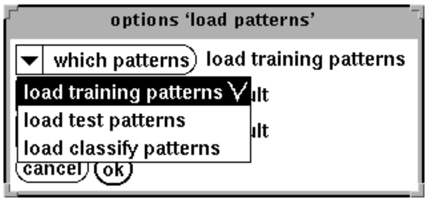


Figure 9.6: *Option control window of* CONVIS.

For many of the actions that exist in neural network simulation programs, initiation often not only involves initializing some parameter values, but it can also require setting some boolean values. In CONVIS, these are called settings. The specification of a setting is done via the following syntax:

```
setting:  setting '{' name state '}'
```

9.2.5 An example: specification of an action `learn`

As an example, consider the specification of an action called `learn`. In this example, the action implements a backpropagation training algorithm.

```
action "learn"  { 1
#                name       type  value min max editable string_value
 parameter {   "nr epochs"  int    1000  0  MAX    1     ""         }
 parameter {  "error_crit"  double 0.05  0 1000    1     ""         }
 parameter {   "save file"  file      4  0    0    1     "bp.net"   }
 parameter { "save cycles"  int    1000  0  MAX    1     ""         }
 parameter {    "momentum"  double  0.1  0   10    1     ""         }
 parameter {       "lrate"  double 0.05  0    1    1     ""         }

 variable  {     "pattern"  int     0    0  MAX    1     "epoch"    }
 variable  {       "epoch"  int     0    0  MAX    1     "epoch"    }
 variable  {       "error"  double  1    0    1    1     "epoch"    }
#                 name                 type  dim monitorable timeid  }
 data      {              "weights" float []        1     ""      }
 data      {          "activations" float []        1     ""      }
 data      {               "biases" float []        1     "epoch" }
 data      {  "train input patterns" float []        1     ""      }
 data      { "train target patterns" float []        1     ""      }
 data      {   "test input patterns" float []        1     ""      }
 data      {  "test target patterns" float []        1     ""      }
 data      {  "learn classification" int   []        1     "epoch" }
 data      { "recall classification" int   []        1     "epoch" }
 data      {       "learn confusion" int   []        1     ""      }
 data      {      "recall confusion" int   []        1     ""      }

 option    { "update"              2 0  "every pattern" "every epoch" }
 option    { "pattern sequence"    2 0  "random"        "round robin" }
 option    { "activation function"  2 0  "sigmoid"       "tanh"        }

 setting   { "classify while learning" 0 }
 setting   { "show confusion matrix"   0 }
 setting   { "autosave"                0 }

}
```

Parameters for this action are the error criterion and the number of epochs after which training has to stop, the name of the file to use for saving a network after every `save cycles` epochs, the momentum determining the influence of previous weight updates and the learning rate determining the speed of learning. Variables are the current pattern and epoch number, and the error generated by the network for all patterns.

Each implemented neural network from the PREENS algorithm library is able to classify a train or test data set during the training process. Datastructures to store the associated data are the input and target patterns, the resulting classifications and confusion matrices. Via the options, it can be decided to perform weight updates after each pattern, or once per epoch. Patterns can be submitted in random order or in a cyclic way. And one out of two activation functions may be selected. The booleans indicating whether the network has to be saved, whether classification of train or test patterns has to be performed or whether a confusion matrix has to be computed can be set via the settings.

When parsing such a specification of an action, a neural network simulation program builds up the `CAction` data structure. In the next section it will be explained how CONVIS and associated tools uses this concept to be able to request and update objects and attributes from a running simulation program.

9.3 PREENS interface definitions

PREENS is a neurosimulator comprising three components:

1. A (expandable) set of executable neural network simulation programs, using the concept of actions and program descriptions explained in the previous section.
2. A (expandable) set of tools for visualization and monitoring of neural network objects, and for dynamic I/O.
3. The graphical user-interface CONVIS, managing a running neural network simulation program and a set of running tools. Via CONVIS, it is possible to request and update objects and attributes from the simulation program.

PREENS was introduced in the introduction of this thesis. For convenience, the global architecture of PREENS is depicted again here:

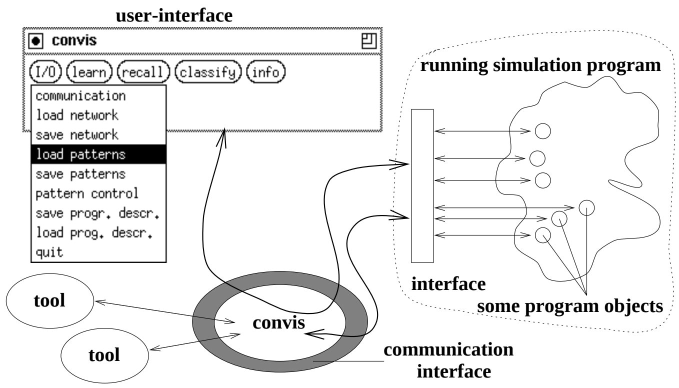


Figure 9.7: *The* PREENS *neurosimulator environment.*

In this environment, a simulation program, CONVIS, and tools run as separate processes coupled via a communication interface using TCP/IP. Note that any of these can be run on different machines, with a different architecture. For example, CONVIS can run on a X-Windows workstation, the simulation program on a parallel processor system, and tools on other machines in a heterogeneous computer network. Values of parameters, settings and options can be requested and updated by CONVIS (if they are specified in the program description). The same can be done with values of data and variables, but furthermore, data and variables can also be "coupled" to input and output tools. As an example, consider the monitoring of the error generated by the neural network for a test set during training.

Via the "edit variable window", a monitoring tool can be coupled to the variable called `error` as follows (see also Figure 9.8:

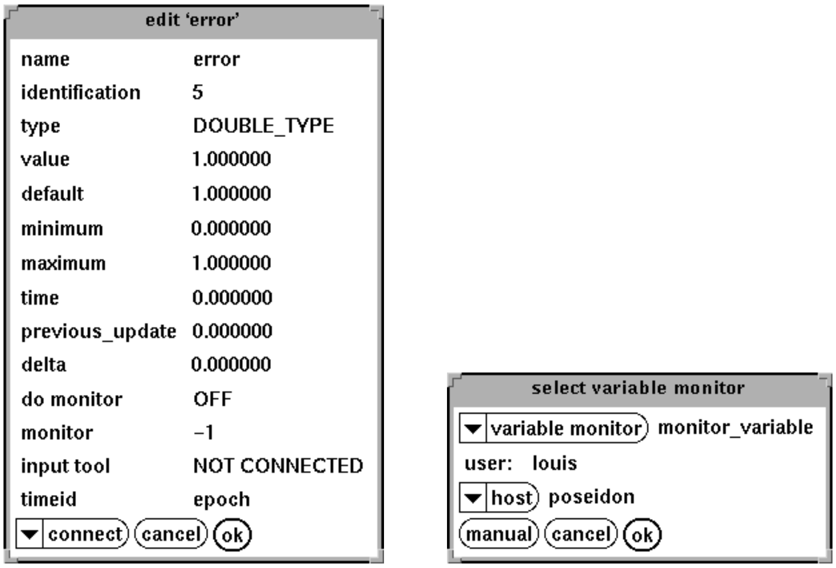


Figure 9.8: *The edit variable window and monitor variable selection window. Via the pull-down menu button* `connect`*, the latter window can be activated. In this window, the tool and machine on which it has to be run can be selected.*

When a tool is coupled to a variable or data object, some fields in the corresponding data structure are set. These fields are:

```
double prev_update;  /* time of previous update of the object */
double delta_t;      /* time amount after which object has to be updated */
int inputtool;       /* identification of input tool coupled to it */
int monitor;         /* identification of output tool coupled to it */
```

The field `delta_time` indicates after how many time steps, the object has to be monitored. For example, if a large test set has to be classified during training, this will take a considerable amount of time, whereas it is often not required to compute the classification at each time step. Setting `delta_time` to, say, 10, indicates that after every 10 steps, the classification has to be computed.

CONVIS maintains an internal process administration of all running tools. Via the fields `inputtool` and `monitor` (which is an output tool), it can get to the required items like the communication channel via which the tool is connected, and the machine type of the machine it is running on.

9.3.1 Interface between CONVIS and a simulation program

In the subsequent text, we call a *programmer* someone who implements neural network simulation programs or tools and wants to interface them to CONVIS. A *user* is someone who uses PREENS. In order to couple a simulation program to CONVIS, a programmer has to perform the following steps:

1. Specify the program description of the program.

2. Write a main program `sim.c`:

   ```
   /* specification of actions here */
   int main (int argc, char **argv)
   {
     FillInSimActions(argc,argv,actions);
     SetUpCommunications();
     ConvisMainLoop(argc,argv);
     return 0;
   }
   ```

   The array `actions` contains a specification of the actions implemented by the program. This array must match the program description.

3. For each action in the program description, implement it following the instructions below.

In the `ConvisMainLoop`, *sim* waits for commands from CONVIS. Four kinds of commands can be issued 1) requests, 2) update, 3) control and 4) interrupt. CONVIS can send requests for the number of actions and their names, the number of parameters or variables of an action, and for each component of an action, its constituents. For each component of an action, CONVIS can send updates. Control commands are RUN_ACTION and CONTINUE_ACTION. Interrupt commands are issued if an event at CONVIS has occurred during the execution of an action. Events are that a user has hit a `halt` or `cancel` button, or that data or a variable from an input tool was received by CONVIS. The simulation program can also generate interrupts, i.e. it can indicate that some data or a variable is ready to be monitored, or that an action has finished.

9.3.2 The action control protocol

Associated with any command, is the identification of the action that is controlled, identification of the component of an action, and information about the contents of any data to follow. In the interface between CONVIS and a simulation program, a command is defined as:

```
typedef struct {
     int command;       /* request, update, control, interrupt  */
     int action_id;     /* action identification               */
     int component_id;  /* component identification            */
     int size;          /* size of any data to follow           */
     double value;      /* to be used for various goals         */
} CCommand;
```

The simulation program can be in two states. When it is running, only interrupt commands are sent by CONVIS. In this state, a RUN or CONTINUE command was previously received from CONVIS, and the `ConvisMainLoop` determined the `action_id` to start up the associated action. If an action is not running, the simulation program is in `ConvisMainLoop`, waiting for request, update, or control commands.

The routine `action_control()` performs four tasks: 1) it detects whether any I/O interrupts arrive, 2) it detects whether it was interrupted by a HALT or CANCEL command, 3) it adjusts the time of each of its variables and data objects, and 4) it checks whether any variables or data have to be monitored.

9.3.3 Accessing components of an action

Until now, interfacing a simulation program with CONVIS has been very simple. Very little has to be changed to the code of existing programs. If programs are written from scratch, the specification of actions with corresponding components can even help in the process of software design. However, the most difficult part of the interfacing procedure is to allow CONVIS to access the neural network attributes. As mentioned before, the actions are contained in an array of `CAction`. For each request or update, a component can be accessed as:

```
actions[action_id].component[component_id].value
```

However, in most neural network simulation programs, objects are accessed by name. Consider for example changing the learning rate, or some weight value:

```
lrate = new_value;
weight[i][j] += lrate*delta[i]*activation[j];
```

This would be implemented as:

```
actions[LEARN].parameters[LRATE].value = new_value;
((double **)actions[LEARN].data[WEIGHT].dataptr)[i][j] +=
       actions[LEARN].parameters[LRATE].value
     * ((double *)actions[LEARN].data[DELTA].dataptr)[i]
     * ((double *)actions[LEARN].data[ACTIVATION]).dataptr)[j];
```

Obviously, this straight-forward implementation introduces a lot of overhead. Selecting and indexing in hierarchical datastructures can be expensive, as stated before. And in order to allow a programmer to write his programs as he is used to, and to restrict the number of efforts required to interface a program to CONVIS, some means has to be defined which maps a name to some data or variable in the array of `actions`.

A set of convenience macro definitions is made available for these purposes:

```
#define p_val(type,id,ix)  ((type)actions[id].parameters[ix].value)
#define v_val(type,id,ix)  ((type)actions[id].variables[ix].value)
#define v_adr(id,ix)       (actions[id].variables[ix].value)
#define o_val(id,ix)       (actions[id].options[ix].state)
#define s_val(id,ix)       (actions[id].settings.states[ix])
```

The code below depicts how these macros can be used in an action `learn`:

```
int learn (CAction *action, int run)
{
   int epoch = v_val(int,LEARN,EPOCH);
   double error = v_val(double,LEARN,EPOCH);
   int nepochs = p_val(int,LEARN,NEPOCHS);
   int npatterns = p_val(int,LEARN,NPATTERNS);
   double errcrit = p_val(double,LEARN,ERRCRIT);

   if (!start(run,0,action))
      return 0;
   while (action_control(action) && epoch<nepochs &&error>errcrit) {
      error = 0.0;
      for (p=0;p<npatterns;p++)
         error += train_pattern(...);
      v_adr(LEARN,EPOCH) = ++epoch; /* make new values available for CONVIS */
      v_adr(LEARN,ERROR) = error;
   }
   return finish(action,error<=errcrit);
}
```

9.3.4 Accessing data

Data can be accessed via the field `dataptr`. This is a `void` pointer to any data defined by a programmer. Using the `CData_Description`, it can be found out how the data is arranged. A convenience routine `AssignData()` is provided for a programmer to specify how his data is arranged:

```
int AssignData (void *dataptr, int id, int index, int dim
      , int d0, int d1, int d2, int d3, int length, int flat)
{
      CData_Description *d = &actions[id].data[ix].description;

      actions[id].data[ix].dataptr = dataptr;
      d->dim = dim;
      d->d[0] = d0; d->d[1] = d1; d->d[2] = d2; d->d[3] = d3;
      d->length = length;
      d->flat = flat;
}
```

As an example of how programmer-defined data can be described, consider the computation of a *confusion_matrix*. During recall, a neural network computes a certain output for each input it receives. A confusion matrix is a way of validating how well the network performs on a set of so-called supervised data, i.e. a set of patterns for which it is known in advance which output the network has to produce. Each row and column of the matrix contain one entry for each of the possible *classes* contained in the data set. The percentage of patterns belonging to class i and which the network has classified as class j is contained in `matrix[i][j]`. In order for a tool to compute the confusion matrix, it has to know for each pattern p the class $class_p$ and $computed_p$. Assume that a programmer wants to equip his program with a datastructure `confusion_matrix`, which contains two arrays of integers, one containing $class_p$, the other $computed_p$. In his program, both arrays are computed as:

```
int confusion_matrix[NPATTERNS][2];

for (p=0;p<NPATTERNS;p++) {
    confusion_matrix[p][0] = targets[p];
    confusion_matrix[p][1] = compute_class(inputs[p]);
}
```

In order to tell CONVIS how the data is structured, the programmer has to specify the confusion matrix in the program description, and use `AssignData()` to specify how the data is arranged:

```
data { "confusion_matrix" int [] 1 "epoch" } /* in program description */
AssignData((void *)confusion_matrix,LEARN,CONFUSION_MATRIX,1
      ,2,0,0,0,NPATTERNS,0);
```

Once this is done, the programmer can refer to the confusion matrix by name, while at any moment, the data can be accessed as specified by its `CData_Description`.

9.3.5 Exotic or distributed data

If a program contains data structured other than multi-dimensional arrays, the mechanism described above cannot be used. This also holds for data distributed over e.g., a network of workstations or a transputer network. This is because the `CData_Description` is not suited for specifying such data. A method is designed to be able to cope with this problem.

The description of data contains a field `data_format`, which can be 1) PREENS_FRMT, 2) USER_DEFINED and 3) DISTRIBUTED. In the first case, the interface as described above can be used. In the other two cases, where the data has some "exotic" format, the data has to be transformed to PREENS_FRMT and vice versa. Figure 9.9 depicts how this is done.

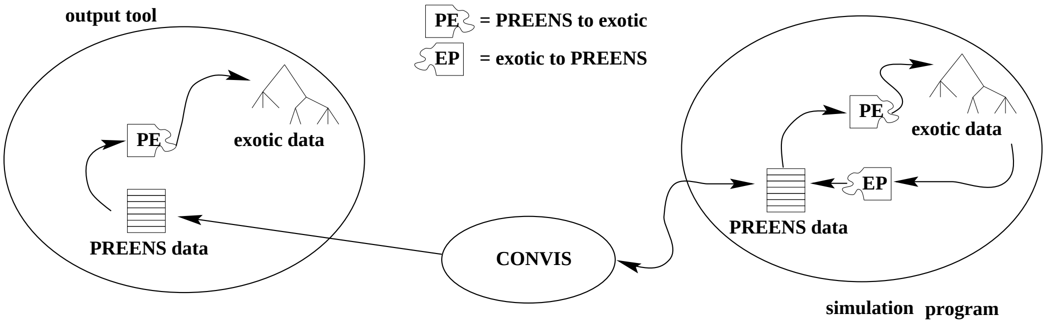


Figure 9.9: *Transforming exotic data to* PREENS *and vice versa.*

If data has to be monitored by some output tool, the simulation program *sim* first checks if the data has to be transformed to PREENS. After transforming, it can be communicated with CONVIS via the communication interface. If an output tool receives data from CONVIS, it checks if the data must be transformed to its exotic format.

In the graduation work of Eric Boon [9], this concept is used for the implementation of an interface to a multi-transputer system. The communication primitives discussed in Chapter 5 are used as a foundation for this parallel interface. For dataset decomposed neural networks, no transformation is required, as on each processor the same copy of the neural network is present. After each epoch, an updated neural network is available at the master processor, so request commands can be satisfied through communication to the master only. Interrupts, control, and update commands are broadcast through the transputer network.

For network decomposed neural networks, two situations may exist, 1) the amount of data requested or to be updated is small enough to be held on one processor, or 2) the amount cannot be stored on one processor. In the first case, for request commands, a distributed data structure is gathered at the master processor, after which it is sent to CONVIS using the normal procedure. For updates, the data is sent to the master processor and subsequently distributed over the processor network.

For the second case, in [9] some modifications are made to the interface between CONVIS and the parallel interface. To be able to select part of a large amount of data, a module called `distribute` is added. This module builds up an administration of the distributed data, based on the PREENS format. This means that given a `CData_Description` and its distributed administration, the module is able to locate where a certain part of the data is located. Using the concept worked out by Boon, instead of requesting or updating a complete data structure, now only part of the data can be accessed using indices and offsets in the `struct_descr` field of the `CData_Description`. However, only preliminary tests are carried out, and this aspect of PREENS must be developed and tested more elaborately.

9.3.6 Interface between CONVIS and tools.

Tools can be coupled to variables and data, using the mechanism as depicted in Figure 9.8. The process administration maintained by CONVIS contains for each tool the communication channels via which it is connected to CONVIS. In the structure describing a variable or data, the identification of the tool which is coupled to it is contained. As explained in the action control protocol 9.3.2, the simulation program *sim* checks for each variable and data object, if it has to be monitored:

```
if (v[i].do_monitor&&v[i].time-v[i].prev_update>=v[i].delta_t)
     SendVariable(v);
```

CONVIS installs an interrupt handler and detects that a variable has to be monitored. Subsequently, it sends the variable to the tool determined from its process administration. All output tools can be implemented using the following set up:

```
#include <preens_convis.h>

int main (int argc, char *argv[])
{
     fd = setup_client_connection(argv[1],atoi(argv[2]));
     ReceiveVariable(&v,fd);
     InitializeTool(&v);
     while (ReceiveVariable(&v,fd))
          HandleVariable(&v);
}
```

The current set of tools implemented is given in Table 9.3. They are described elaborately in [115].

confusion_matrix_plotter	-	For graphical visualization of confusion matrices
convis_matlab	-	Interface between CONVIS and matlab [66]
convis_xv	-	Interface between CONVIS and xv [10]
data_tool	-	A general data evaluation tool, containing basic statistical analysis features, like standard deviation, min, mean, max
kohonen_projection	-	For visualizing the classification results of a Kohonen map
monitor_variable	-	Monitoring values of a variable per time step
monitor_data	-	Monitoring values of individual data elements per time step
plot_image	-	Visualizing images.
save_data	-	Tool to save data in PREENS format.

Table 9.3: *List of tools contained in* PREENS.

9.4 An example: training remotely sensed data

In [89, 87], the application of neural networks to the classification of remotely sensed imagery is introduced. At the Institute for Remote Sensing Applications from the JRC in Ispra, Italy, research is carried out for the automated classification of satellite images. Various types of imagery are available, like radar (SAR), Landsat-TM and Spot. The goal is to use these images and classify each pixel's ground cover class. The work described here was performed in close cooperation with Ron Schoenmakers from the JRC.

9.4.1 Initiation phase

In [87], the classification performance of a neural network for combined six-band Landsat-TM and one-band ERS-1/SAR PRI imagery from the same scene is presented. Different combinations of the data, either raw, segmented or filtered, using the available ground truth polygons, training and test sets are created. The training sets are used for learning while the test sets are used for verification of the network. The combination of two types of imagery offers the possibility to test their influence on the classification accuracy. From the scene used, the Lisbon area in Portugal, two images are contained, one Landsat-TM recorded June 24, 1991 and one ERS-1/SAR PRI image from March 21 1992. The imagery is geo-referenced and re-sampled into pixels with 25 meter ground resolution. The error caused by the geo-referencing is within one pixel. The image size is 2518 columns by 2363 rows. The original 16-bit per pixel ERS-1 image is scaled to one byte-per-pixel. From the Lisbon area, ground truth data (collected in June 1991) are available for 9 classes:

1 sand	(825,798)	2 grassland	(3358,3165)	3 water	(4726,3878)
4 cereals	(369,73)	5 forest	(719,71)	6 urban	(353,298)
7 vineyard	(168,33)	8 marshland	(356,123)	9 aquatic vegetation	(262,259)

Table 9.4: *The 9 ground truth classes. In parenthesis, the number of pixels per class for the train set and test set.*

The ground truth data is divided into a training set (11163 pixels) for training and a test set (8698 pixels) for testing. The satellite images contain a number of *bands*, each band representing a range in the visible, reflected infra-red (IR), thermal IR or microwave portions of the electro-magnetic spectrum. Using the raw one-band radar, the six-band optical and the seven-band combined data, processed by segmentation and filtering techniques, eight different data sets are produced:

1	1-band SAR, un-filtered
2	1-band SAR, filtered
3	6-band optical, raw
4	6-band optical, segmented
5	7-band optical + SAR, raw
6	7-band optical + SAR, filtered
7	7-band optical + SAR, segmented
8	7-band optical + SAR, filtered and segmented

Table 9.5: *The eight data sets used for this experiment.*

The segmentation method used [88] is a combination of an improved edge detection method and a region growing method. It is beyond this thesis to explain more about this process, but the idea is that through segmentation, neighboring pixels having similar spectral values are clustered in one segment. In the segmented data sets used here, the average of each band of a segment is assigned to all pixels it contains. So all pixels in one segment have identical spectral values. The data extracted from the SAR image is speckle filtered by the method described in [73]. For the 1-band SAR data no segmentation was performed, as the average classification performance (ACP) was low for the speckle filtered image, and segmentation was not expected to improve the ACP significantly.

In the initiation phase, the task to be performed, the data sets, and the neural network model to be used are identified. The first two items are described above. Using PREENS, several tests were performed with neural networks from the PREENS algorithm library:

artmap	-	Implementation of several ART networks by Parcival Willems [120].
backprop	-	Implementation of backpropagation neural network with momentum training.
boltzmann	-	Implementation of a variant of the Boltzmann neural network as described in [55].
fieldnet	-	Implementation of a the fieldnet neural network as described in [85].
kohonen	-	Implementation of the Kohonen SOM.
counterprop	-	Implementation of the counterpropagation neural network [44].

Table 9.6: *Neural networks from the* PREENS *algorithm library*

From both the training sets and the test sets, random selections were made of 500 patterns. Using these sets, PREENS was used to quickly evaluate the classification performance of each of the neural networks in the algorithm library. The backpropagation neural network outperformed the rest. It was already shown in [53, 52] that the multi-layered perceptron can be used for this application. The tool that was particularly used for this initial examination was the confusion matrix monitoring tool. Both the classification performance for the training set and the test set can be monitored during training using this tool (see Figure 9.10).

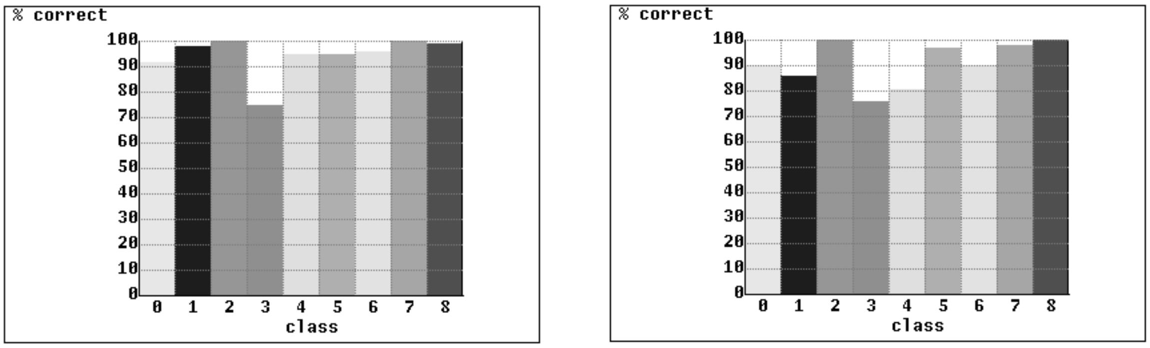


Figure 9.10: *Values of diagonal of confusion matrix with average classification results. The left image is the diagonal for the training set, the right image is the diagonal for the test set.*

In general, classification performance in remote sensing is expressed as the number of pixels correctly classified, divided by the total number of pixels [11]. In [87], for the evaluation of classification performance, the averages over classes are used. This was done to eliminate differences between the number of pixels available per class. For example, a neural network may classify a scene of pixels containing water with high percentage correct, whereas it is unable to classify vineyards. Because of the unequal distribution of pixels between these classes (see Table 9.4), the overall average classification performance would be biased by the large amount of pixels in the class of water.

9.4.2 Tuning and testing phases

In the tuning phase, the optimal initial parameters for the backpropagation neural network (backprop), and its architecture are determined. Numerous experiments using PREENS were performed, training neural networks with the different training sets and comparing the classification results of the test sets. The following observations were made:

1. Small initial weight values are required, with $w_{ij} \in [-0.15 \cdots 0.15]$

2. The learning rate must be low, $\epsilon = 0.001$, which confirms to the rule of thumb $\epsilon \approx 10/p$, with p the number of patterns. For higher learning rates, the network fluctuates heavily between different weight states.

3. Training must be online, i.e. for each pattern, the weights of backprop are updated. For off-line training, no networks were able to learn the training set. Apparently, updating weights per epoch averages the weight changes and the network gets easily trapped in local minima. With online training, this effect is avoided.

4. The `tanh` activation function gives better results than the traditional `sigmoid`. Furthermore, the networks converge faster. The reason why this is the case is not known.

5. Scale the patterns within a small interval, say $[-0.9, .9]$. For both the sigmoid and `tanh` activation functions, the active domain is near zero. This means that in that range, small changes in the net-input will result in relatively large changes in the activation output. If the patterns are not scaled, i.e., they contain pixel values which are between $[0 \cdots 255]$, the net-input of the activation functions is in the domain at which they are not very active.

For training the data sets, two 4-layered neural network architectures were used: a 6x26x18x9 and a 7x30x20x9 network. By monitoring the ACP during training, it showed that even after many training iterations (> 5000), for most data sets, no loss in performance was observed. However, for the segmented data sets such a loss was observed ,— a phenomenon called over-training —, where the neural network becomes too much specialized for the training set. For smaller sized neural networks, or a smaller number of iterations, a better ACP was achieved. This indicates that the segmented data is easier to learn. Results for the 8 data sets are depicted in Table 9.7:

	raw	fil	seg	fil+seg
SAR	28.7	34.3		
optical	92.2		90.8	
combined	92.1	93.0	92.0	82.6

Table 9.7: *ACP (in %) for each of the 8 experiments.*

It appears that just the SAR data is not enough to distinguish correctly between different ground cover classes. On the other hand, it seems that using both the optical and SAR imagery in combination with segmentation and filtering techniques produces no significant increase in ACP. Combining filtering and segmentation results in a drop in ACP.

To get an insight in how the different imagery, filtering and segmentation influence the classification performance, a further investigation in how the different ground cover classes are recognized is required. Table 9.8 depicts for each experiment the ACP for each of the 9 classes.

	6raw	6seg	7raw	7fil	7seg	7fil+seg
1	91.6	93.2	92.4	92.5	94.5	92.0
2	84.8	96.7	85.4	85.7	84.0	88.4
3	100	100	100	100	100	99.8
4	86.3	69.4	90.4	97.3	98.6	94.5
5	76.1	76.1	78.9	80.3	67.6	99.0
6	99.7	99.0	90.3	92.6	99.0	86.9
7	97.0	100	97.0	93.9	87.9	6.1
8	98.4	100	98.4	98.4	100	100
9	100	100	100	100	100	99.2
avg.	92.2	90.8	92.1	93.0	92.0	82.6

Table 9.8: *ACP (%) for 6 and 7 band imagery.*

9.4.3 Classification

For testing PREENS in an operational phase, CONVIS was coupled to an input tool, reading images in `.lan` format and producing data in the form of an input pattern containing 6 bands:

```
data { "classify patterns" float 1 [width*height] 6 0 "epoch" }
```

Furthermore, CONVIS was coupled to the output tool `convis_xv`. This tool starts up the image visualization program `xv`, with the option to poll for creation or update events of a file. For each classified image CONVIS receives from the simulation program, it transfers the data to the output tool. This tool overwrites the file, after which `xv` displays the new image.

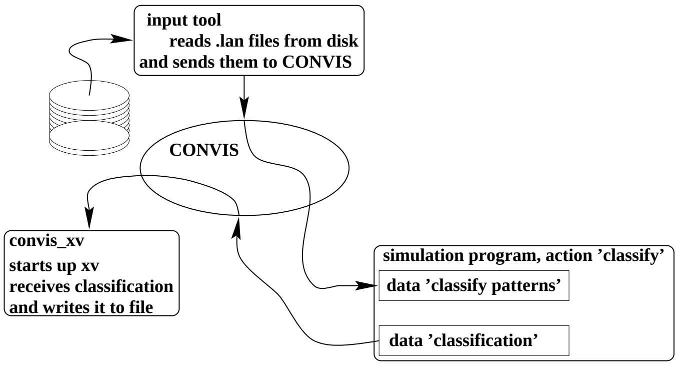


Figure 9.11: *Set up of the classification phase.*

For each image to be classified, the classification results are stored in a file. The corresponding parameter `savefile` is set via the action control menu of the action “classify”.

It was discussed that in an operational phase, the neural network part of the neurosimulator may be extracted for a stand-alone end-application running without the neurosimulator. In such a set up, the input tool, the tuned neural network, and the code for saving the classification results would be extracted and integrated into the end-application. In the work of Schreurs and Hendriks [91], it was examined how PREENS neural network code can be extracted and embedded into a stand-alone application. Their conclusion was that it is relatively easy to do this. The actions `load_network` and `classify` must be extracted, and a set of macro-definitions must be used which mirror the way the components of the neural network are accessed in CONVIS.

9.4.4 Conclusions

Based on the observations made when considering features of existing neurosimulators, and considering the requirements specified by users and environments in the world of neurocomputing, three design criteria for PREENS were identified. Following these criteria, the design, implementation and application of PREENS is presented in this chapter. Using the concept of action-oriented *program descriptions*, CONVIS can relatively easy be interfaced with neural network simulation programs. Using the communication and data interface, tools can be added to the PREENS tool library. As few assumptions are made about how a neural network is implemented, the network simulation code can be tailored to the needs of a specific application or neural network model, and can thus be highly efficient and compact. CONVIS can operate in parallel with tools and the simulation program it is connected to. Because of this exploitation of the computing resources in a processor network, PREENS runs faster and can handle a larger amount of data than current neurosimulators. The design of PREENS makes it suitable for appliers of neural networks to use it as an efficient platform to solve their applications. It can also very well be used as an extendible general purpose neurosimulator for controlling simulations written by neural network programmers.

PREENS was tested with neural network simulation programs running on various execution platforms. For controlling dataset parallel implementations of simulations running on MIMD-parallel systems, CONVIS is very well suited. For network decomposed simulations, a preliminary interface was tested using an administration of distributed data structures.

10

Conclusions

Outline

This chapter summarizes the findings about platforms for artificial neural network simulations. In the first part of this thesis, the suitability of MIMD-parallel execution platforms for artificial neural networks is examined. In particular, a performance prediction method is introduced and evaluated for transputer-based implementations of backpropagation and Kohonen neural networks.

In the second part of this thesis, neurosimulators as a platform for simulating neural networks are discussed. The features of the new, action-oriented neurosimulator called PREENS are discussed here.

Introduction: platforms for artificial neural networks

In this thesis, the reader is made familiar with several operational aspects from the world of neurocomputing. In Chapter 2, an introduction is given to these aspects. Artificial neural networks, high-performance platforms for executing them, and a performance prediction method for examining the suitability of these platforms are introduced. Furthermore, a taxonomy of users "doing neurocomputing" is given, the typical phases occurring in the neurocomputing life-cycle are identified, and an introduction to PREENS, a platform for simulating neural networks, is given. In this chapter, conclusions about these aspects are drawn and some final remarks are made.

MIMD-execution platforms and the transputer

In Chapter 3, the architecture of MIMD-parallel computers and in particular multi-transputer systems is presented. The processor architecture and communication networks of three multi-transputer systems, the Nijmegen Super Cluster (NSC), the GCEL-512 (GCEL) and the PowerXPlorer (PX) are discussed. Aspects of how to program these systems using the operating systems Helios (for the NSC) and Parix (for the GCEL and PX) are mentioned. It was concluded that at present, the T8xxx transputer is more or less outdated because of its low processor performance. Current parallel processor systems like the PowerXPlorer tend to use faster, more up-to-date CPUs as the basic processing element. However, the techniques discussed in this thesis are not restricted to transputer systems only; they are valid for any MIMD-architecture were there exist a number of nodes connected via a communication network.

Performance prediction

In Chapter 4, it is concluded that there are a number of disadvantages when using the techniques of performance benchmarking and performance modeling. With performance benchmarking, a large number of criteria have to be taken into account before a benchmark can be used to predict the performance of an execution platform for an application. It was argued that performance measures like MCUPS are meaningless, completely useless performance scales, unless it is exactly stated what such a parameter implies. For example, *[Rumelhart,momentum,batch-update,6x20x30x9] backprop* MCUPS is a more descriptive performance measure than just plain MCUPS. Considering performance modeling, it was also concluded that a large number of parameters influence the performance. For example, the hardware architecture, operating system, compilers, compiler options, run-time libraries, implementation aspects, and the size and complexity of the application, are factors that have an effect on the performance that can be achieved. If all of these factors are kept constant, it may be possible to come up with reliable estimates. However, if any of them is changed, a new estimation has to be made. Because of the problems with performance benchmarking and modeling, in this thesis, a new technique is introduced which combines the two techniques.

The combined method of performance modeling and kernel benchmarking presented in this thesis defines the total required calculation and communication time as Equation (10.1):

$$\begin{aligned} T_{calc}(P,n,w,p) &= \frac{1}{P} \cdot \sum_{i=1}^{n_f} (N_i(n,w,p) \cdot t_i) \\ T_{comm}(P,n,w,p) &= C(P,n,w,p) \cdot t_{comm} \\ T_{total}(P,n,w,p) &= T_{calc}(P,n,w,p) + T_{comm}(P,n,w,p) \end{aligned} \tag{10.1}$$

In this method, a program is modeled in terms of a number of n_f function kernels. Kernel benchmarks are executed on one single processor to measure the time t_i required for executing each kernel i. The $N_i(n,w,p)$ represents the number of times each kernel is called and is based on the number of neurons n, the number of weights w and the number of patterns p. The communication time is modeled as the number of times a single information unit (e.g., a connection or activation value) must be sent over a physical communication link, $C(P, n, w, p)$, multiplied by the time required for communicating one value, t_{comm}.

A communication layer

In Chapter 5, a communication layer implementing the typical patterns of communication required for MIMD-parallel neural network simulations is presented and analyzed. Algorithms for *broadcast*, *gather* and *accumulate* are discussed for MIMD-computer networks organized in grid and tree architectures. The communication complexity for these algorithms is given by:

$$\begin{aligned}
T^{grid_broadcast}(P,s) &= (width(P) + height(P) - 2) \cdot s \cdot t_{comm} \\
T^{tree_broadcast}(P,s) &= 3 \cdot depth(P) \cdot s \cdot t_{comm} \\
T^{grid_gather_accumulate}(P,w) &= (width(P) + height(P) - 2) \cdot w \cdot (t_{comm} + t_{acc}) \\
T^{tree_gather_accumulate}(P,w) &= 3 \cdot depth(P) \cdot w \cdot (t_{comm} + t_{acc})
\end{aligned}$$

Using these models, the communication time $T_{comm}(P, s)$ can be predicted for any number of processors P, where it is only required to measure the time t_{comm} over one communication link and the accumulation time t_{acc} on one processor. For all three transputer systems used, the NSC, the GCEL and the PX, this method is validated quantitatively. For a varying number of processors P, and several message sizes s, measurements and predictions using the method are made. The predictions have an average accuracy within 5%, and all are accurate within ±10%.

Dataset decomposition

When using dataset decomposition for parallel neural network simulations, a set of p patterns is equally distributed over the available P processors. A complete copy of the neural network runs on each processor. In Chapter 6, this technique is presented and analyzed for the transputer-based implementation of two popular neural network models, backpropagation and Kohonen neural networks. For both neural networks, the kernel functions are determined and measured on one processor. Also the communication requirements are given. Based on these, predictions are made which all have an accuracy within a couple of percentages.

Using the performance prediction method, the suitability of MIMD systems like the transputer can be expressed in terms of MCUPS or ICPS, speedup, efficiency and scalability. It appears that dataset decomposition is a highly efficient technique. This can be concluded from the following observations:

1. The maximal performance as expressed by Equations (6.4) and (6.7) is almost reached.

2. Though the speedup limit is reached for small problem sizes, in general, near linear speedups are reached. Furthermore, for two real applications (see Section 6.7), the speedup limit is reached far beyond the available number of processors (e.g., 194 (PX), 881 (NSC) and 929 (GCEL) for Nettalk on a grid).

3. High scalabilities are achieved and the scalability limit is not reached.

Network decomposition

In Chapter 7, these issues are considered for network decomposition, the collection of decomposition methods for distributing one single neural network over a number of processors. The implementation of backpropagation and Kohonen neural networks via network decomposition is presented and analyzed. A new communication method is described for gathering the distributed activations of the backpropagation neural network. Again, function kernels are determined and measured on one processor. Using the model for calculation and communication times, performance predictions are made with and accuracy within $\pm 15\%$ for backpropagation and $\pm 10\%$ for Kohonen. Compared to the results achieved with the dataset decomposition technique, it appears that network decomposition is less suitable, in particular for backpropagation. This can be concluded based on the following considerations:

1. The achieved performances are less, consider for example Figure 6.9 and Table 7.7 for the Nettalk problem and Figure 6.11 and Table 7.11 for Satdat.

2. The speedup limit is reached relatively soon, as can be seen in Table 7.3.

3. The scalability and efficiency are low, consider Section 7.7.

For the Kohonen SOM, these considerations do not show such a dramatic drop in suitability. The reason is that for KSOMs, the communication requirements are far lower than for network decomposed backpropagation networks.

Neurosimulators

Three classes of neurosimulators are distinguished in Chapter 8: application-oriented, algorithm-oriented, and general programming systems. Users in the world of neurocomputing are: model builders, tool builders, applied researchers and end-users. For some users, one class of neurosimulators may be more suited than for others. In Chapter 8, it is concluded that current neurosimulators are based on a general (hierarchical) neural network datastructure or some neural network description language. This approach requires neural networks to be specified in a prescribed standard manner, which usually means that their implementations are not as efficient as would be the case with implementations specifically designed for a (neural network and) target execution platform. Furthermore, it forces neural network programmers to program their applications following the NDL syntax or using some set of neural network library routines. Furthermore, it appears that neural network simulation programs all have a similar structure, implementing a limited set of actions. And finally, in many cases the neural network code in neurosimulators represents only a small part of the total code required. The larger part is required for implementing user-interface and I/O.

PREENS, an action oriented neurosimulator

Based on these observations, in Chapter 9, a new kind of neurosimulator is presented. PREENS uses a conceptual model of programs rather than of neural networks. In this model, a program is described via the actions that are implemented. In PREENS, the user-interface CONVIS and neural network programs are loosely coupled. Via a program description and a communication interface, CONVIS can control a running simulation program and access its components. It can also exchange components of the neural network with tools from the PREENS algorithm library. As programmers do not have to conform to some general neural network datastructure, the neural network code can be implemented specifically tailored for an application. Note that PREENS fully exploits the availability of workstations in a computer network. As tools, CONVIS and a neural network simulation program can be run on any machine, the environment can be executed in parallel. Therefore, it can operate faster and handle larger amounts of data than current neurosimulators.

Currently, PREENS can handle tools and simulation programs running on workstations and transputer systems. It has successfully been tested on heterogeneous processor networks consisting of Dec, Sun and Hp workstations and a Parsytec transputer system. For dataset decomposed neural network simulation programs, the communication libraries of PREENS operate satisfactory. Preliminary tests indicate that the concept of transforming “exotic” data to and from PREENS format also shows promising opportunities.

Final considerations

The performance prediction method described and used in this thesis can be used to evaluate the suitability of MIMD-execution platforms for artificial neural network simulations. Note that this method is a scalable performance prediction method, because based on the measurements of calculation time on one processor, and on the communication time over one communication link, the performance can be predicted for larger-scaled processor networks. Using the method, predictions are made for the performance of backpropagation and Kohonen neural network implementations. Whereas the predictions are good, in general with an accuracy within $\pm 10\%$, for network decomposed backpropagation networks it appears that transputers do not provide a very efficient execution platform. However, for dataset decomposed implementations, and also for the network decomposed Kohonen network, good suitability criteria are achieved.

This indicates that for MIMD platforms, when implementing a neural network, it should be considered whether dataset decomposition is feasible. If this is not the case, only small processor networks, possibly with a large amount of memory per node (e.g., 64 Mb compared to 4 Mb for the T8xxx transputer), may be used to implement a neural network. This observation matches current trends in MIMD-processor technology, where individual processing elements are becoming more powerful.

The performance prediction method and the concept of action-oriented program descriptions described in this thesis could also be used for other application areas:

- For any MIMD-parallel implementation of an application, the total execution time can be expressed as the sum of the times required for calculation and for communication. To make this feasible, it is required to identify the kernel benchmarks, and measure these and the basic communication time t_{comm}. Equation (10.1) can then be used to determine the total exection time, where instead of the number of neurons, weights and patterns (n, w, p), the parameters corresponding to the application have to be filled in. Note that for implementations that exploit techniques for overlapping communication and calculation, a more complicated model is required.
- The concept of action-oriented program descriptions could also facilitate a means of designing tool boxes for application areas like, e.g., image processing. Similar to neural network applications, with image processing a relatively small set of actions can be identified, such as loading and storing the image, filtering operations, edge detection, segmentation, etcetera. The possibility of incorporating these within one tool-set and exploiting the resources in a distributed environment by executing them on different processors would justify the use of an approach like the one sketched in this thesis for PREENS.

Bibliography

[1] G.M. Amdahl. Validity of the single processor approach to achieving large scale computer capabilities. In *Proc. AFIPS Spring Joint Comp. Conf. 30*, pages 483–485, Atlantic City (NJ), 1967.

[2] M. Azema-Barac, M. Hewetson, M. Recce, J. Taylor, P. Treleavan, and M. Vellasco. Pygmalion Neural Network Programming Environment. In B. Angeniol and B. Widrow, editors, *International Neural Network Conference*, pages 1237–1244, Paris, July 1990. Kluwer Academic Publishers.

[3] G. Bader and B. Przywara. T9000 - A Preliminary Evaluation of Arithmetic Performance. Brief Note, Interdisziplinares Zentrum fur Wissenschaftliches Rechnen, Universitat Heidelberg, Im Neuenheimer Feld 368, D-6900 Heidelberg, iwr1.iwr.Uni-Heidelberg.de, May 1993.

[4] V. Barbosa and P.M. Lima. On the Distributed Parallel Simulation of Hopfield's Neural Networks. *Software-Practice and Experience*, 20(10):967–983, October 1990.

[5] M. Berry, G. Cybenko, and J. Larson. Scientific Benchmark Characterizations. *Parallel Computing*, 17(2):1173–1194, 1991.

[6] D.P. Bertsekas and J.N. Tsitsklis. *Parallel and distributed computation.* Prentice Hall, 1989.

[7] Silvio Bierman and Hans van Hooft. Kohosim, a parallel simulation environment for kohonen neural networks. Under graduate project, 1996.

[8] C.M. Bishop. *Neural networks for pattern recognition.* Clarendon Press, Oxford, 1995.

[9] Eric Boon. The P in preens, A dataset parallel interface for convis and experiences with a parallel counterpropagation simulator. Master's thesis, University of Nijmegen, Computer Science, June 1997. thesisnr 415.

[10] John Bradley. Xv, an interactive image manipulation program for the x windows system, 1994. Version 3.10a.

[11] J.B. Campbell. *introduction to remote sensing.* The Guilford Press, New York, 1996.

[12] G.A. Carpenter and S. Grossberg. Art 2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26(23):4919–4930, December 1987.

[13] G.A. Carpenter and S. Grossberg. A massively parallel architecture for a self-organizing neural pattern recognition machine. *Computer Vision, Graphics, and Image Processing*, 37:54–115, 1987.

[14] G.A. Carpenter, S. Grossberg, and J.H. Reynolds. Artmap: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network. *Neural Networks*, 4:565–588, November 1991.

[15] G.A. Carpenter, S. Grossberg, and D.B. Rosen. Fuzzy art: Fast stable learning and categorization of analog patterns of an adaptive resonance system. *Neural Networks*, 4:759–771, June 1991.

[16] Edingburgh Parallel Computing Centre. Annual Report and Project Directory, 1990-1991.

[17] G. Chinn, K.A. Grasjki, C. Chen, C. Kuszmaul, and S. Tomboulian. Systolic array implementation of neural nets on the maspar mp-1 massively parallel processor. In *Proceedings of the IJCNN-90 II*, pages 169–173, San Diego, 1990.

[18] L-C. Chu and B.W. Wah. Optimal Mapping of Neural Network Learning on Message-Passing Multicomputers. *Journal of Parallel and Distributed Computing*, 14:319–339, 1992.

[19] L.J. Clarke. Tiny version 1.0, discussion and user guide. Technical report, Edinburgh Parallel Computing Centre, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, May 1989.

[20] M. Cosnard, J.C. Mignot, and H. Paugam-Moisy. Implementations of Multilayer Neural Networks on Parallel Architectures. In 2^{nd} *International Specialist Seminar on Parallel Digital Processors*, Lisbonne, Portugal, april 1991.

[21] H.J. Curnow and B.A. Wichmann. A Synthetic Benchmark. *Comput. J.*, 19(1):43–49, 1991.

[22] A. d'Acierno, R. Del Balio, and R. Vaccaro. Fully connected neural networks: simulation on massively parallel computers. *Artificial Neural Networks*, pages 1489–1500, 1991.

[23] Perihelion Software Ltd. (Ian Davies). *The Helios Parallel Operating System*. Prentice Hall, 1991.

[24] K.M. Dixit. The SPEC Benchmarks. *Parallel Computing*, 17(2):1195–1209, 1991.

[25] J. Dongarra and W. Gentzsch. Benchmarking of High Performance Computers. *Parallel Computing*, 17(2):1067–1069, 1991.

[26] J.J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee, Computer Science Department, Knoxville, TN 37996-1301, September 1992.

[27] M. Duranton, F. Aglan, and N. Mauduit. Hardware Accelerators for Neural Networks: Simulations in Parallel Machines. In D. Heidrich and J.C. Grossetie, editors, *Computing with T-Node Parallel Architecture*, pages 235–264. ECSC, EEC, EAEC, Brussels and Luxembourg, 1991.

[28] W. Eppler, M. Rinderspacher, and M. Rudolph. A Digital Signal Processor for Simulating Backpropagation Neural Networks. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 1565–1568. Elsevier Science Publishers (North-Holland), 1991.

[29] D. Ercoşkun and K. Oflazer. Experiments with Parallel Backpropagation on a Hypercube Parallel Processor System. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 1465–1468. Elsevier Science Publishers (North-Holland), 1991.

[30] A. Zell *et al.* *SNNS Stuttgart Neural Network Simulator.* University of Stuttgart, IPVR, Breitwiesenstrasse 20–22, 70565 Stuttgart Germany, 1994. User Manual, Version 3.3 Report 3/94 (revised).

[31] B.W. Char *et al.* *Maple V Language Reference Manual.* Springer Verlag, 1991.

[32] S.E. Fahlman. The Recurrent Cascade-Correlation Architecture. Technical Report CMU-CS-91-100, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, May 1991.

[33] S.E. Fahlman and C. Lebiere. The Cascade-Correlation Learning Architecture. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*. Morgan Kaufmann, 1990.

[34] J.A. Feldman, M.A. Fanty, N.H. Goddard, and K.J. Lynne. Computing with Structured Connectionist Networks. *Communications of the ACM*, 31(2):170–187, Februari 1988.

[35] H.P. Flatt and K. Kennedy. Performance of Parallel Processors. *Parallel Computing*, 12:1–20, 1989.

[36] B.M. Forrest, D. Roweth, N. Stroud, D.J. Wallace, and G.V. Wilson. Implementing Neural Network Models on Parallel Computers. *The Computer Journal*, 30(5):413–419, 1987.

[37] G.Fox, M.Johnson, G.Lyzenga, S.Otto, J.Salmon, and D.Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, New Jersey, 1988.

[38] J. Ghosh and K. Hwang. Mapping Neural Networks onto Message-Passing Multicomputers. *Journal of Parallel and Distributed Computing*, 6:291–330, 1989.

[39] Parsytec Gmbh. *Supercluster technical documentation.* Parsytec Gmbh, Juelicher Straße 338 D-5100 Aachen Germany, April 1989.

[40] N.H. Goddard, K.J. Lynne, T. Mintz, and L. Bukys. Rochester Connectionist Simulator. Technical Report 233 (revised), University of Rochester, Computer Science Department, Rochester, New York 14627, October 1989.

[41] K.F. Goser. Challenge of ANN to Microelectronics. In S. Gielen and B. Kappen, editors, *ICANN '93*, pages 1025–1029. Springer-Verlag, 1993.

[42] J.L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[43] I.S. Han, K.H. Ahn, T.H. Park, and K.H. Jun. Adaptable VLSI Neural Network of Tens of Thousand Connections. In I. Aleksander and J. Taylor, editors, *Artificial Neural Networks 2*, pages 1423–1426. Elsevier Science Publishers (North-Holland), 1992.

[44] R. Hecht-Nielsen. Applications of Counterpropagation Networks. *Neural Networks*, 1:131–139, 1988.

[45] J.N.H. Heemskerk. *Neurocomputers for Brain-Style Processing. Design, Implementation, and Application.* PhD thesis, Leiden University, Dept. of Experimental Psychology, 1995.

[46] A.J.G. Hey. The Genesis Distributed Memory Benchmarks. *Parallel Computing*, 17(2):1275–1283, 1991.

[47] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

[48] R.W. Hockney and C.R. Jesshope. *Parallel computers : architecture, programming and algorithms.* Hilger, 1988.

[49] J.J. Hopfield. Networks and Physical Systems with Emergent Collective Computational Abilities. In *Proc. Natl. Acad. Sci. USA 79*, pages 2554–2558, 1982.

[50] Motorola Inc. IBM Microelectronics. *PowerPC 601, RISC Microprocessor User's Manual.* IBM Microelectronics, Motorola Inc., IBM Microelectronics, Mail Stop A25/862-1, PowerPC Marketing, 1000 River Street, Essex Junction, VT 05452-4299, 1993.

[51] INMOS International. *OCCAM 2 reference manual.* Prentice Hall Internationa, 1988.

[52] I. Kanellopoulos, A. Varfis, G.G. Wilkinson, and J. Megier. Land cover discrimination in spot hrv imagery using an artificial neural network. *International Journal of Remote Sensing*, 13(5):917–924, 1992.

[53] I. Kanellopoulos and G.G. Wilkinson. Experiments with backpropagation neural networks for image classification. Technical Report I.91.119, JRC, Image Processing Lab, I-21020, Ispra (VA) Italy, 1990.

[54] K. Kant. *Introduction to Computer System Performance Evaluation.* Computer Science Series. McGraw-Hill International Editions, 1992.

[55] B. Kappen. Using Boltzmann Machines for probability estimation. In S. Gielen and B. Kappen, editors, *ICANN '93*, pages 521–526. Springer-Verlag, September 1993.

[56] Bert Kappen. Personal communication, during several meetings at the Biophysiscs Lab in Nijmegen.

[57] C. Kelley and T. Williams. *Gnuplot Unix version 3.5, online manual.* info-gnuplot dartmouth.edu, 1986–1993.

[58] T. Kohonen. *Self-Organization and Associative Memory.* Springer Verlag, Berlin, second edition, 1988.

[59] T. Kohonen, J. Kangas, and J. Laaksonen. Som pak, the self-organizing program package. Technical report, Laboratory of Computer and Information Science, Rakentajanaukio 2 C, SF-o2150 Espoo, Finland, November 1992. Version 1.2.

[60] T. Kohonen, J. Kangas, J. Laaksonen, and K. Torkkola. Lvq pak, the learning vector quantization package. Technical report, Laboratory of Computer and Information Science, Rakentajanaukio 2 C, SF-o2150 Espoo, Finland, October 1992. Version 2.1.

[61] M. Korsloot, A.J. Klaassen, and J.M. Mulder. The Suitability of Transputer Networks for Various Classes of Algorithms. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, chapter 15, pages 275–291. Steven Ericsson Zenith., July 1989.

[62] R. Leenders. EENS, an Execution Environment for Neural Systems. Master's thesis, Nr. 139, University of Nijmegen, Faculty of Mathematics and Informatics, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, January 1990.

[63] R.R. Leighton. The aspirin/migraines neural network software. Technical Report MP-91W00050, MITRE Washington Neural Network Group, The MITRE Corporation Washington C^3I Division 7525 Colshire Drive McLean, Virginia 22102, October 1992. User's Manual Release V6.0.

[64] B. Levin. Implementation of the SANS Algorithm on the CM2. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 1473–1476. Elsevier Science Publishers (North-Holland), 1991.

[65] Inmos Limited. Transputer Reference Manual. Prentice Hall, 1988.

[66] Mathworks, Cochituate Place 24, Prime Pathway, Natic, Mas 01760. *Matlab, High Performance Numeric Computation and Visualization Software*, 1992. Version 4.0 User's Guide.

[67] Inmos/SGS-Thomson Microelectronics. The T9000 Transputer Product Overview, 1991.

[68] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J. Beer. The Ring Array Processor: a Multiprocessing Peripheral for Connectionist Applications. *Journal of Parallel and Distributed Computing*, 14:248–259, 1992.

[69] J.M.J. Murre. Transputer Implementations of Neural Networks: an Analysis. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 1537–1540. Elsevier Science Publishers (North-Holland), 1991.

[70] J.M.J. Murre and S.E. Kleynenberg. The metanet network environment for the development of modular neural networks. In B. Angeniol and B. Widrow, editors, *Proceedings of the International Neural Network Conference*, pages 717–720, Paris, July 1990. Kluwer Academic Publishers.

[71] J.M.J. Murre, R.H. Phaf, and G. Wolters. CALM networks: a modular approach to supervised and unsupervised learning. In *Proceedings of the IJCNN*, pages 649–656, Washington, 1989.

[72] Inc. NeuralwareWare. Neuralworks professional ii. 202 Park West Drive, Pittsburgh PA 15275.

[73] E. Nezry, H-G. Kohl, and H. de Groof. Restoration of Textural Properties in SAR Images using Second Order Statistics. In *Proceedings of the International Geoscience and Remote Sensing Symposium (IGARSS94)*, pages 2165–2167, Pasadena, California, USA, August 8-12 1994.

[74] K. Obermayer, H.Heller, H. Ritter, and K. Schulten. Simulation of Self-Organizing Neural Nets: a Comparison between a Transputer Ring and a Connection Machine CM-2. In *Proceedings of the Third Conference of NATUG*, Sunnyvale, CA, 1990.

[75] K. Obermayer, H. Ritter, and K. Schulten. Large-Scale Simulations of Selforganizing Neural Networks on Parallel Computers: Application to Biological Modelling. *Parallel Computing*, 14:381–404, 1990.

[76] Paugam-Moisy. A Spy of Parallel Neural Networks. Technical Report TR 90-27, Ecole Normale Supérieure de Lyon, IMAG, Lyon, France, 1990.

[77] M. Plonski. Rcs, genesis, and sfinx:three public domain simulators for neural networks. *Neural Network Review 4*, 1990. paper obtained via Neural Network Archive http://www.lpac.ac.uk/SEL-HPC/Articles/NeuralArchive.html.

[78] AFCEA International Press. DARPA Neural Network Study, November 1988.

[79] U. Ramacher, W. Raan, J. Anlauf, J. Beichter, U. Hachmann, N. Brühls, M. Weßeling, and E.Sicheneder. Multiprocessor and Memory Architecture of the Neurocomputer SYNAPSE-1. In S. Gielen and B. Kappen, editors, *ICANN '93*, pages 1034–1039. Springer-Verlag, 1993.

[80] M.L. Recce, P.V. Rocha, and P.C. Treleaven. Neural Network Programming Environments. In I. Aleksander and J. Taylor, editors, *Artificial Neural Networks 2*, pages 1237–1244. Elsevier Science Publishers, September 1992.

[81] G.D. Richards. The Implementation of Backpropagation on a Transputer Array. In J. Kerridge, editor, *Developments using Occam*, pages 173–179, 1988.

[82] G.D. Richards and T. Tollenaere. Documentation for Rhwydwaith Version 2.1. Technical Report ECSP-UG-7, University of Edinburgh, July 1989.

[83] D.E. Rumelhart and J.L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, 1986.

[84] A.J.M. Russel. Simulating Neural Networks on a Multi-Transputer System. Master's thesis, Nr. 175, University of Nijmegen, Faculty of Mathematics and Informatics, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, January 1991.

[85] A.J.M. Russel and Th.E. Schouten. FIELDNET, a Dynamic Network for Pattern Classification. In S. Gielen and B. Kappen, editors, *ICANN '93*, pages 456–459. Springer-Verlag, September 1993.

[86] Mimetics S.A. Mimenice artificial neural networks simulator, user manual. Avenue Sully-Prudhomme, 92298 Chatenay-Malabry, France.

[87] Ron Schoenmakers and Louis Vuurpijl. Segmentation and classification of combined optical and radar imagery. In *Proceedings of IGARSS'95*, Florence, September 1995.

[88] R.P.H.M. Schoenmakers. *Segmentation of Remotely Sensed Imagery.* PhD thesis, Informatica, Katholieke Universiteit Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, September 1995.

[89] R.P.H.M. Schoenmakers, G.G. Wilkinson, and Th.E. Schouten. Results of a hybrid segmentation method. In *SPIE*, Rome, September 1994. SPIE.

[90] Th.E. Schouten. Internal report. Faculty of Mathematics and Informatics University of Nijmegen The Netherlands, 1991.

[91] R. Schreurs and C. Hendriks. Extracting PREENS Neural Network Code and Embedding it into a Stand-alone Simulation Program. Technical report, University of Nijmegen, Faculty of Mathematics and Informatics, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, October 1993.

[92] T.J. Sejnowski, P.K. Kienker, and G.E. Hinton. Learning Symmetry Groups with Hidden Units: Beyond the Perceptron. *Physica*, 22D:260–275, 1986.

[93] T.J. Sejnowski and C. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, pages 145–168, 1987.

[94] A. Singer. Exploiting the Inherent Parallelism of Artificial Neural Networks to Achieve 1300 Million Interconnects per Second. In B. Angeniol and B. Widrow, editors, *Proceedings of the International Neural Network Conference*, pages 656–660, Paris, July 1990. Kluwer Academic Publishers.

[95] A. Singer. Implementations of Artificial Neural Networks on the Connection Machine. *Parallel Computing*, 14:305–315, 1990.

[96] G. Smith. Backpropagation with Dynamic Topology and Simple Activation Functions. Technical Report TR 90-12, School of Information Science and Technology, Discipline of Computer Science Flinders University of South Australia PO Box 2100, Adelaide, SA, 5001, 1992.

[97] California Scientific Software. Brainmaker. 10024 Newtown Road, Nevada City, CA 95959.

[98] H. Speckmann, P. Thole, and W. Rosenstiel. COKOS: a COprocessor for KOhonen's Selforganizing Map. In S. Gielen and B. Kappen, editors, *ICANN '93*, pages 1040–1044. Springer-Verlag, 1993.

[99] J.B. Theeten, M. Duranton, N. Maudit, and J.A. Sirat. The LNeuro-Chip: A Digital VLSI with On-Chip Learning Mechanism. In B. Angeniol and B. Widrow, editors, *Proceedings of the International Neural Network Conference*, pages 593–596. Kluwer Academic Publishers, July 1990.

[100] T. Tollenaere and G.A. Orban. Simulating Modular Neural Networks on Message-Passing Multiprocessors. *Parallel Computing*, 17(1):361–379, 1991.

[101] T. Tollenaere and G.A. Orban. Transparent Problem Decomposition and Mapping - a CSTools based Implementation. In P. Welch, editor, *Transputing '91*, pages 107–123, 1992.

[102] T. Tollenaere, M.M. van Hulle, and G.A. Orban. Parallel Implementation and Capabilities of Entropy Driven Artificial Neural Networks. *Journal of Parallel and Distributed Computing*, March 1992. excact reference to be found (pp).

[103] P.C. Treleaven. Neurocomputers. *International Journal of Neurocomputing*, 1:4–31, 1989.

[104] A. Ultsch and H.P. Siemon. Exploratory Data Analysis: Using Kohonen Networks on Transputers. Technical Report Bericht Nr. 329, University of Dortmund, Fachbereich Informatik, Postfach 500500, D-4600 Dortmund 50, December 1989.

[105] A.J. van der Steen. The Benchmark of the EuroBen Group. *Parallel Computing*, 17(2):1211–1221, 1991.

[106] F.A. van Schaik and J.A.G. Nijhuis. Introduction to neural network design with nnsim 3.0. Technical Report IMS-TB-05/89, Institution for Microelectronics, Allmandring 30, 7000 Stuttgart 80, Germany, November 1989.

[107] J. Vanhala and K. Kaski. Simulating Neural Networks in Distributed Environments. In J. Wexler, editor, *Developing Transputer Applications (Proceedings OUG 11)*, pages 129–141, Edinburgh, Scotland, September 1989. IOS.

[108] Bart Veer. *The CDL Guide*. Helios Technical Guides. Distributed Software Ltd., 670 Aztec West, Bristol BS12 4SD, UK, January 1990.

[109] L.G. Vuurpijl and Th.E. Schouten. Control and Visualization of Neural Networks in the PREENS Project. In *Proceedings of the third workshop of the Esprit Parallel Computing Action*, Bonn, May 1991.

[110] L.G. Vuurpijl and Th.E. Schouten. Suitability of Transputers for Neural Network Simulations. In W. Joosen and E. Milgrom, editors, *Parallel Computing: From Theory to Sound Practice*, pages 528–537. IOS Press, 1992.

[111] L.G. Vuurpijl and Th.E. Schouten. Performance of the GCel-512 for Parallel Neural Network Simulations. In *End report of the CAMPP '93 programme (University of Amsterdam and Parsytec Gmbh)*. University of Nijmegen, 1993.

[112] L.G. Vuurpijl and Th.E. Schouten. PREENS, a Parallel Research Execution Environment for Neural Systems. In *High Performance Computing and Networking '94*, München, April 1994.

[113] L.G. Vuurpijl and Th.E. Schouten. A Scalable Performance Prediction Model for Parallel Neural Network Simulations. In *High Performance Computing and Networking '94*, München, April 1994.

[114] L.G. Vuurpijl, Th.E. Schouten, and J. Vytopil. Performance Prediction of Large MIMD Systems for Parallel Neural Network Simulations. *Future Generation Computing Systems*, 11(2):221–232, 1995.

[115] Louis Vuurpijl. Preens tutorial, how to use tools and nn simulations. Technical report, University of Nijmegn, November 1995.

[116] Louis Vuurpijl, Theo Schouten, and Jan Vytopil. Convis: Action oriented control and visualization of neural networks. Technical report, University of Nijmegen, Faculty of Mathematics and Informatics, Informatics for Technical Applications, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, November 1994.

[117] R. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1023, October 1984.

[118] R.P. Weicker. A Detailed Look at some Popular Benchmarks. *Parallel Computing*, 17(2):1153–1172, 1991.

[119] G. Whittington and C.T. Spracklen. A structured design, development and integration methodology for real-world applications of artificial neural networks. In I. Aleksander and J. Taylor, editors, *Artificial Neural Networks 2*, pages 1245–1251. Elsevier Science Publishers, September 1992.

[120] P. Willems. The ART Neural Networks Enlightened: Implementation on Sequential and Parallel Computer Systems. Master's thesis, University of Nijmegen, Faculty of Mathematics and Informatics, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, September 1993.

[121] S.A. Williams. *Programming Models for Parallel Systems.* John Wiley & Sons Ltd., 1990.

[122] M. Wilson and J.M. Bower *et al.* *Genesis and Xodus Documentation.* Caltex, ftp://ftp.bbb.caltech.edu, Februari 1991.

[123] M.A. Wilson, U.S. Bhalla, J.D. Uhley, and J.M. Bower. Genesis: A System for Simulating Neural Networks. In D.S. Touretzky, editor, *Advances in Neural and Information Processing Systems*, pages 485–492. Morgan Kaufmann, 1989.

[124] M. Witbrock and M. Zagha. An Implementation of Backpropagation Learning on GF11, a Large SIMD Parallel Computer. *Parallel Computing*, 14:329–346, 1990.

[125] C-H. Wu, R.E. Hodges, and C-J. Wang. Parallelizing the Self-Organizing Feature Map on Multiprocessor Systems. *Parallel Computing*, 17(1):821–832, 1991.

[126] H. Yoon and J.H. Nang. Multilayer Neural Networks on Distributed Memory Multiprocessors. In B. Angeniol and B. Widrow, editors, *Proceedings of the International Neural Network Conference*, pages 669–672, Paris, July 1990. Kluwer Academic Publishers.

[127] X. Zhang, M. McKenna, J.P. Mesirov, and D.L. Waltz. The Backpropagation Algorithm on Grid and Hypercube Architectures. *Parallel Computing*, 14:317–327, 1990.

Samenvatting

Deze dissertatie is het resultaat van het PREENS onderzoeksprojekt, verricht bij de onderzoekslijn EITA van de faculteit Wiskunde en Informatica aan de Katholieke Universiteit Nijmegen. In dit proefschrift worden twee omgevingen belicht die gebruikt worden voor kunstmatige neurale netwerken: *neurosimulatoren*, dat wil zeggen: software omgevingen voor het simuleren van neurale netwerken en MIMD-*parallelle computers*, krachtige machines voor de snelle uitvoering van de rekenprocessen in neurosimulatoren.

Parallellisme en neurale netwerken

Parallellisme komt overal voor. In de natuur, waar bijvoorbeeld ontelbare mieren tegelijkertijd aan een termietenhoop bouwen. Maar ook in het dagelijks leven, waar machines parallel aan een taak werken, waar bouwvakkers met z'n allen een huis bouwen en waar voertuigen tegelijkertijd in het verkeer deelnemen.

Ook kunstmatige neurale netwerken vertonen parallellisme op verschillende niveaus. Op een gedetailleerd niveau bevatten ze een groot aantal rekeneenheden, de *neuronen*, die met elkaar zijn verbonden via een nog groter aantal axonen en synapsen (gemodelleerd door *gewichten*). In de biologie werken neuronen, axonen en synapsen parallel in de tijd samen. Op een hoger niveau komen verschillende modules van lagen van neuronen en gewichten voor, die alle parallel ten opzichte van elkaar werken.

Parallelle computers lijken een voor de hand liggend platform te zijn om kunstmatige neurale netwerken op te implementeren, vanwege het parallellisme dat deze bevatten. Het voordeel is dat zulke implementaties sneller uitgevoerd kunnen worden dan op sequentiële computers. Het nadeel is dat parallelle systemen moeilijker zijn te programmeren en meestal specifiek gericht zijn op één neural network model of één bepaalde toepassing. In de literatuur zijn een groot aantal parallelle implementaties van kunstmatige neurale netwerken te vinden. Op een laag niveau van parallellisme worden een-op-een afbeeldingen van gewichten en neuronen op analoge of digitale chips gebouwd. Een overzicht van dit soort "neuro-asics[1]" wordt gegeven in [45, 79, 103]. Een breed spectrum van andere parallelle platforms zoals de Connection Machine [94], GF11 [124], MasPar [17] en transputersystemen [81, 82, 84, 104, 107] zijn gebruikt voor hogere (meer abstracte) niveaus van

[1] asic = application specific integrated circuit

parallellisme. Deze machines zijn ofwel massief parallel (bevatten veel, relatief simpele, processoren), of ze bevatten een kleiner aantal meer algemeen toepasbare processoren.

In dit proefschrift worden transputersystemen gebruikt voor de implementatie van neurale netwerken. Met name wordt een model geïntroduceerd waarmee de geschiktheid van de transputer voor neurale netwerken kan worden bepaald. Een transputersysteem behoort tot de klasse van MIMD[2]-systemen, computers bestaande uit meerdere processoren die verbonden zijn in een communicatienetwerk. In tegenstelling tot massief parallelle computers bevatten MIMD-systemen slechts een beperkt aantal processoren. Dit heeft als gevolg dat het aantal neuronen en gewichten het aantal processoren ver overtreft. Dus als een neuraal netwerk op een MIMD-systeem wordt geïmplementeerd, dan moeten het benodigde geheugen en de rekentaken verdeeld worden over de beschikbare processoren. Technieken voor het opdelen van een neural netwerk op MIMD-systemen zoals de transputer zijn beschreven in bijvoorbeeld [38, 100, 110] en worden in detail behandeld in de hoofdstukken 5, 6 en 7. Twee principes zijn hierbij van belang: zorg ervoor dat iedere processor evenveel werk te doen heeft en zorg ervoor dat de hoeveelheid rekenwerk die met informatieuitwisseling, synchronisatie en communicatie te maken heeft zoveel mogelijk beperkt blijft.

In hoofdstuk 3 wordt de architektuur belicht van MIMD-systemen en in het bijzonder van drie transputersystemen, het Nijmegen SuperCluster, de GCEL-512 en de PowerXplorer. Er wordt behandeld op welke wijze deze systemen te programmeren zijn en hoe er gebruik gemaakt kan worden van de besturingssystemen Helios en Parix. De technieken die in dit proefschrift worden beschreven kunnen voor iedere MIMD-architektuur worden gebruikt, dus niet alleen voor transputersystemen.

Voorspellen van de prestatie

Bij het bepalen van de geschiktheid van een computersysteem voor een bepaalde toepassing, kunnen verschillende vragen gesteld worden. De eerste is welke rekensnelheid behaald kan worden, gegeven een bepaalde machineconfiguratie en type en grootte van de toepassing. De tweede vraag is of door de communicatie- en rekencapaciteit te vergroten, de totale rekentijd verlaagd kan worden, oftewel welke prestatieverbetering of “speedup” behaald kan worden. De derde vraag betreft de schaalbaarheid van het computersysteem en de toepassing, oftewel blijft de rekentijd konstant als de grootte van zowel het computersysteem als de toepassing gelijkmatig opgeschaald wordt. In hoofdstuk 4 wordt een methode geïntroduceerd die de prestatie van MIMD-systemen voor kunstmatige neurale netwerken kan voorspellen. Volgens deze methode wordt voor een gegeven systeem de rekentijd en communicatietijd gemodelleerd. Op basis van de gemeten rekentijd op één processor en de gemeten communicatietijd tussen twee processoren kunnen voorspellingen gedaan worden voor grotere processorsystemen. Met behulp van deze methode kunnen de drie vragen die hierboven zijn gesteld zeer goed worden beantwoord.

[2] MIMD staat voor “multiple instruction multiple data”. Zo bestaan er ook SIMD-systemen, waarbij de “s” staat voor “single”.

Zoals beschreven in [110, 113], vereisen neurale netwerk implementaties op parallelle systemen verschillende typische soorten van communicatie. In hoofdstuk 5 wordt een communicatielaag geïntroduceerd die deze soorten implementeert. Algorithmen voor *broadcast*, *gather* en *accumulate* communicaties worden behandeld en de bijbehorende modellen voor de vereiste communicatietijd worden uitgewerkt in dit hoofdstuk. Met behulp van deze modellen kan de communicatietijd $T_{comm}(P, s)$ voorspeld worden voor ieder aantal processoren P en iedere boodschapgrootte s. Voor de drie gebruikte transputersystemen, blijken de voorspelde waardes een precisie van rond de 5% te hebben en accuraat te zijn binnen 10%.

Het modelleren van de rekentijd gaat volgens een nieuwe methode van *kernel* benchmarking en performance modeling. In hoofdstuk 4 worden een aantal nadelen van het gebruik van klassieke performance benchmarking en performance modeling methoden genoemd. Een performance schaal zoals MCUPS is nutteloos, tenzij precies wordt aangegeven wat nu precies tot zo'n getal heeft bijgedragen, zoals "*[Rumelhart,momentum,batch-update,6x20x30x9] backprop* MCUPS". Volgens de nieuwe methode wordt de rekentijd gemodelleerd als:

$$T_{calc}(P, n, w, p) \quad = \quad \frac{1}{P} \cdot \sum_{i=1}^{n_f} (N_i(n, w, p) \cdot t_i) \qquad (10.2)$$

De rekentijd van een programma wordt hierbij gemodelleerd als de som van de benodigde rekentijden voor n_f rekenfuncties, waarvan voor elk de rekentijd wordt gemeten op slechts een processor. Deze methode noemen we ook wel "kernel benchmarking".

Datasetdecompositie en netwerkdecompositie

De hierboven beschreven methode om de prestatie van een parallel systeem te voorspellen is in dit proefschrift toegepast voor twee technieken, datasetdecompositie en netwerkdecompositie. Bij de eerste techniek wordt een verzameling van p door het neurale netwerk te leren patronen gelijkmatig verdeeld over P processoren. Op iedere processor draait een identieke kopie van het neurale netwerk. In hoofdstuk 6 wordt deze techniek geëvalueerd voor twee typen neurale netwerken, backpropagation en Kohonen. Voor beide modellen zijn kernel benchmarks gedefiniëerd en gemeten op de drie verschillende transputersystemen. Samen met de modellen voor de communicatietijd kunnen voorspellingen worden gemaakt voor de totale rekentijd op een willekeurig transputernetwerk. De geschiktheid van een MIMD-systeem zoals de transputer kan vervolgens uitgedrukt worden in termen van MCUPS of ICPS, speedup, efficiency en scalability. De resultaten gepresenteerd in hoofdstuk 6 geven aan dat datasetdecompositie een bijzonder geschikte techniek is voor MIMD-systemen.

In hoofdstuk 7 wordt netwerkdecompositie behandeld, een verzameling technieken om een neuraal netwerk over meerdere processoren te verdelen. De voorspellingsmethode wordt getoetst voor netwerkdecompositie via de implementatie van backpropagation en Kohonen netwerken. Wederom worden kernel benchmarks gedefiniëerd en gemeten op een processor. Voorspelde waardes wijken maximaal 15% af voor backpropagation en 10% voor Kohonen.

Vergeleken met de resultaten behaald met datasetdecompositie, blijkt netwerkdecompositie minder geschikt te zijn, in het bijzonder voor backpropagation vanwege de benodigde hoeveelheid informatieuitwisseling tussen processoren.

Neurosimulators

In dit proefschrift wordt een overzicht gegeven van de typische gebruikers van neurosimulatoren en de activiteiten die zij uitvoeren. Er zijn gebruikers die modellen bouwen, die "tools[3]" bouwen, die neurale netwerken gebruiken voor een toepasssing, en eindgebruikers. In hoofdstuk 8 worden de typische componenten van neurosimulatoren beschreven. De meeste neurosimulatoren zijn gebaseerd op een hiërarchische datastruktuur of een netwerkbeschrijvingstaal. Dit vereist dat neurale netwerken op een voorbeschreven manier ontwikkeld moeten worden, wat inhoudt dat de resulterende implementaties niet zo efficiënt zijn als die welke specifiek voor een neuraal netwerk en/of computersysteem ontwikkeld zijn. Verder blijkt dat de meeste neurale netwerkprogrammatuur een identieke struktuur heeft en slechts een beperkt aantal acties ondersteunt. Tenslotte blijkt dat het grootste gedeelte van de code van een neurosimulator bestaat uit de user-interface en I/O voor visualisatie en lezen en schrijven van en naar de harddisk.

PREENS

De in dit proefschrift geïntroduceerde neurosimulator PREENS is gebaseerd op een conceptueel model van programma's in plaats van neurale netwerken. In dit model worden de acties die het *programma* implementeert beschreven volgens een zogenaamde actiegeoriënteerde programmabeschrijving. Een algemene user-interface, CONVIS, beheert een op deze manier beschreven neurale netwerksimulatie, en kan tevens de data in het programma manipuleren en uitwisselen met tools. Daar er geen rekening hoeft te worden gehouden met een algemene datastruktuur of netwerkbeschrijvingstaal, kan een neuraal netwerkprogramma op maat geïmplementeerd worden voor een bepaalde toepassing. De verzameling componenten bestaande uit CONVIS, een neuraal netwerkprogramma en additionele tools kan op een netwerk van computers uitgevoerd worden. Hierdoor kan de PREENS neurosimulator sneller zijn en meer data verwerken dan andere neurosimulatoren.

Het concept van PREENS is geëvalueerd voor verschillende toepassingen, waaronder satellietbeeldherkenning. PREENS is getest op heterogene computersystemen bestaande uit Unix werkstations en transputernetwerken. Voor parallelle implementaties voldoet de PREENS communicatielaag in het geval van datasetdecompositie, terwijl voor netwerkdecompositie het concept van transformeren van "exotische" data van en naar het PREENS formaat een geschikte oplossing lijkt te zijn.

[3]Tools zijn softwareprogrammas, gereedschappen, die gebruikt kunnen worden voor een bepaald doel, zoals het visualiseren van informatie in een kunstmatig neuraal netwerk.

Slotoverwegingen

De ontwikkelde methode om de rekenprestatie van MIMD-computersystemen te voorspellen voor een toepassing is gebruikt om de geschiktheid van dit soort systemen voor kunstmatige neurale netwerken te bepalen. De in dit proefschrift gemelde resultaten geven aan dat redelijk nauwkeurige (in het algemeen binnen 10%) voorspellingen kunnen worden gedaan. Voor backpropragation neurale netwerken die geïmplementeerd zijn via netwerkdecompositie blijken transputersystemen geen efficiënt executieplatform. Maar voor datasetdecompositie en ook voor Kohonen netwerken die zijn geïmplementeerd via netwerkdecompositie blijken transputersystemen wel geschikt.

Dit geeft aan dat voor de implementatie van neurale netwerken op dit soort systemen de mogelijkheid om datasetdecompositie te gebruiken nader moet worden bekeken. Als er geen efficiënte methoden van datasetdecompositie worden gevonden, dan zullen slechts kleinere systemen gebruikt mogen worden, met eventueel meer geheugen per processor. Deze observatie komt overeen met de huidige trends in de MIMD-processortechnologie, waar individuele processoren steeds krachtiger worden.

De in dit proefschrift beschreven methode om de prestatie van een MIMD-computersysteem te voorspellen en het concept van op acties gebaseerde programmabeschrijvingen kunnen ook gebruikt worden voor andere toepassingsgebieden:

◇ Voor iedere MIMD-parallelle implementatie van een toepassing kan de totale uitvoeringstijd worden uitgedrukt in termen van reken- en communicatie-tijden. Via het definiëren en meten van kernel benchmarks en de communicatietijd, kan de geschiktheid van het MIMD-computersysteem worden voorspeld op het gebied van rekenkracht, speedup, efficiency en scalability.

◇ Het concept van op actie geörienteerde programmabeschrijvingen kan ook gebruikt worden om andere softwareomgevingen te ontwikkelen, zoals voor beeldbewerking. Evenals bij neurale netwerktoepassingen, zijn hierbij een beperkt aantal acties te identificeren zoals het laden en bewaren van een beeld, filter-, edge-detectie, segmentatie en andere beeldbewerkingsoperaties. De mogelijkheid om deze te verenigen in een softwareomgeving en gedeeltes op verschillende processoren in een gedistribueerd netwerk te laten executeren, rechtvaardigt een benadering zoals beschreven in dit proefschrift voor PREENS.

Curriculum Vitae

Louis Vuurpijl was born in Gorinchem, the Netherlands on April 4, 1964. In 1982 he passed his exams for the VWO at the Lorentz Scholengemeenschap in Arnhem, and started to study computer science in Nijmegen. He received his M.Sc. degree in computer science in 1989 from the University of Nijmegen.

From 1989 till 1991, Louis worked at the Department of Technical Informatics as a system manager and programmer, to fulfill his alternative military service. During this work he started his job as a PhD student. He finished the PREENS project in 1995, and since then he has been working as a researcher at the NICI, also at the University of Nijmegen. In 1998 he will start a 3-year post-doctoral research at the NICI.