Parallelizing Feed-Forward Artificial Neural Networks on Transputers

Svend Jules Fjerdingstad Carsten Nørskov Greve

19. September 1991

Abstract

This thesis is about parallelizing the training phase of a feed-forward, artificial neural network. More specifically, we develop and analyze a number of parallelizations of the widely used neural net learning algorithm called *back-propagation*.

We describe two different strategies for parallelizing the back-propagation algorithm. A number of parallelizations employing these strategies have been implemented on a system of 48 transputers, permitting us to evaluate and analyze their performances based on the results of actual runs. It should be noted, that we have emphasized the qualitative aspect of the analyses, due to belief that this should be sufficient to allow us to achieve a fair understanding of the factors determining the behaviour of these parallel algorithms. Our main interest is not the theoretical analysis and modeling of the algorithms. Instead, we are more interested in discovering and deling with some of the specific circumstances which have to be considered when a parallelized neural net learning algorithm is to be implemented on a system of transputers. Part of our purpose is to investigate whether it is possible to exploit the computational resources of a transputer system to a degree comparable to what is achieved on other architectures. In this connection we discuss the problems inherent in comparing different parallel neural net simulators, and criticize the most commonly used measurement for evaluating the preformance of a parallelization of the back-propagation algorithm.

It turns out to be very difficult (if not impossible) to give general recommendations as to which algorithm should be perferred. The appropriate choice depends on the specific neural net problem in question.

In addition to the above, it is our intention that it should be possible to use this thesis as a sort of Transputer User's Guide to Parallelizing Feed-Forward Artificial Neural Networks.

Throughout the thesis, we present our own results and to some extent describe the results reported by others in the literature.

Acknowledgments

We would like to express our gratitude to Ole Caparni for his supervision of our work and his many helpful suggestions. Out thanks also go to the University of Odense for making their transputer system available to us.

Contents

Preface

Artificial Intelligence (AI) is a research field that covers various efforts of modeling and/or recreating aspects of netural intelligence. One of the two competing paradigms of the AI community is based on the idea of recreating intelligent behaviour by imitating the architecture of a biological brain. Such imitations are often referred to as artificial neural networks, and the programs used to run them are called neural net simulators.

These networks are not programmed to perform some specific task. Instead they are supposed to be able to learn the given task by a trail-and-error method in which a supervisor supplies the neural network with the correct responses to all inputs. The most widely used learning rule is the so-called *back-propagation* algorithm.

However, traning these artificial neural networks is a computationally very intensive task, requiring millions of floating point multiplications even for small networks and small problems. Moreover, neural nets require large amounts of memory. These two facts make work on neural nets a very time consuming business, putting an effective limit on the size of the problems than can be undertaken.

There are several ways to try to compensate for this disadvantage of the neural networks approach to AI.

One is to reduce the size of the problem by preprocessing the input data, thereby reducing either the number of iterations necessary to train the net or the size of the net itself. Sometimes such reduction of input can be done using some form of decimation to reduce the number of input patterns, or some projection or feature extraction algorithm can be employed to reduce the dimension of individual input patterns. It must be noted, though, that such reductions are almost always problem specific, so that this approach cannot be generalized to cover all kinds of problems.

Another possibility is the attempt of improving the performance of the

back-propagation learning rule, either by *ad hoc* modifications or by applying results from numerical optimization theory. Work on the so-called *conjugate gradient* methods [Johansson] belongs to the latter category.

A third approach is to make existing algorithms run faster either by implementing them directly in hardware (using VLSI techniques [Tank] or optics [Abu-Mostafa]) or by modifying them to run on some parallel architecture of existing processors. It is this latter part of the third approach, that we are going to deal with in this thesis: How to parallelize the back-propagation learning algorithm.

Chapter 1

An Introduction to Artificial Neural Network

1.1 Motivation

For hundreds of millions of years living brains, brought into existence and continually refined by the ever on-going evolutionary processes of natural selsction, were the only devices capable of performing information processing in general.

Then human beings invented the digital computer, an artificial information processing device which introduced the prospect of performing arbitrary computations outside biological nervous systems in human beings and other animals. However, it soon became obvious that in some important respects the properties of digital computers were quite different from those of living brains.

There is a difference in structure: The digital computer (usually) has only one processing unit which is, however, often quite powerful. Brains, on the contrary, consist of densely interconnected neurons working in parallel, each of which is a small and comparatively simple processing unit. With respect to imfprmation processing capabilities, thougth, the structual difference is not the most important one because all computers possess the ablilty of simulating other structures than their own.

More important is the difference in how the abililty to perform some new function is acquired: Brains learn, whereas computers have to be programmed. In order to perform a given task, the digital computer needs

software, programs implementing algorithms that explain in detail how that task may be performed. A computer without a program cannot process information or carry out computations. Traditionally, some human being has both to understand a given information processing function and to devise an algroithm for implementing it before the computer can be programmed to perform that function. There are, however, many tasks for which formal algorithms do not yet exist, or for which it is virtually impossible to write down a series of logical steps that will make the computer arrive at the correct answer. Such tasks usually involve obsering a large number of complex, context dependent rules most of which are as of yet unknown. Examples of such tasks are the complex pattern-recognition problems inherent in understanding continuous speech, indentifying handwritten character, recognizing faces, or providing a spartial interpretation of two-dimensional images.

Often, though, it is possible to specify the task quite accurately by giving a very large set of examples showing how objects in some input space should be associated with objects in some output space. Usually humans are good at learning to perform such tasks because the brain of higer animals has evolved to generalize well when presented with a number of examples. This fact has lead to the development of *Artificial Neural Networks* [Rosenblatt, Rumelhart], devices designed to model the workings of biological neural networks at some level of detail in order to attain (some of) the desirable capabilities of the human brain.[1]

Like the real thing, an artificial neural net is a massively parallel interconnected system of simple processing elements. In this respect, artificial neural nets are based on our present understanding of biological nervous systems. It should be noted, thougt, that the human brain is more complex by several magintudes than any artificial neural network currently existing. It is estimated [Schwartz] that there are on the order of 10^{10} to 10^{11} neurons in the human brain. Each of these neurons typically receives input from thousands or even tens of thousands of synapses connection it to other neurons, and the resulting activity of the neuron can be transmitted through other thousands of synapses to impinging neurons, thereby influencing their future activity.

Also, when speaking of the relation between artificial and biological

[1]Research into this subject (and related subjects) is also known as *Connectionism* (because of the important role played by the connections in the net) or *Parallel Distributed Processing*(PDP)[Rumelhart], although the name *artificial neural nets* apparently has an appealing flavour to it, since this is a very widely used term.

neural networks, it is worth noticing that for several reasons the current level of detail in the modeling of individual neurons is quite coarse. One of the reasons is the somewhat limited knowledge presently available about the physiology of biological neurons. Another important reason is that not all aspects of neuro-physiology may be relevant, if achieving the adaptability of the brain is the primary goal rather than creating a system that models nature as closely as possible.

Contrary to traditional computer systems an artificial neural network is a non-programmed adaptive information processing system that learns through experience. During the learning phase it is presented with a number of examples of how it should behave on some input. Gradually, the neural network adapts itself to the given task through trial-and-error. Instead of being given a sequence of instructions showing how to carry out some function the network is able to generate its own internal rules governing the association between input and output. Those rules are constructed and continually refined by comparing the results produced by the network with the ones found in the examples.

Artificial neural networks consist of a set of simple processing elements called *units* (sometimes also referred to as *neurons* because of the association with biological nervous systems), and a set of *links* connecting these units. Activity spreads through the net from units to units via the links, each of which has a *weight* (or *connection strength*) associated with it. The weight, which determines the amount of effect one unit has on another, is usually represented by a real number. Depending on the sign of the weight the link will be either an *excitatory* or an *inhibitory* connection, i.e. it will increase or decrease the activity of the recipient unit. Input to each unit from the net is formed by combining the output of all units feeding into this unit with the weights of the corresponding connections. The activity of each unit is then determined by applying an *activation function* on the input received and, possibly, the current activity of the unit. Finally, the *output function* maps the activity of the unit to an output signal, which is then propagated through the links as input to other units. Often, though, as will be the case for all the networks in this thesis, this output function is simply the identity function, so that the output from a unit is equal to its activity.

Input from the environment can be impressed on the network by stimulating special units designated for external input, the so-called *input units* (or *sensory* units). Patterns of activity observed on a certain set of units, the so-called *output units* (or *motor* units), are interpreted as the response

of the network to the given input, i.e. the classification of the input pattern proposed by the network.

As can be seen, the response of the network to a given input pattern is determined by the weights of the connections between the units. The function or association computed by the network can therefore be modified by changing these weights. Therefore it is the pattern of connectivity that constitutes what the system knows and determines how it will respond to arbitrary input. But if knowledge resides in the strengths of the connections, then learning must be a matter of finding suitable values for these weights. It follows that a neural net *learning rule* can be formulated as a rule for how weights should be modified in response to incorrect or partially correct output produced by the network.

1.2 The Structure of a Unit

In general, each neural net unit is connected to a number of other units from some of which it receives input. The unit calculates an activity value which is sent to a number of other units in the net. Figure 1.1 is an illustration of a unit with associated input links and output links.

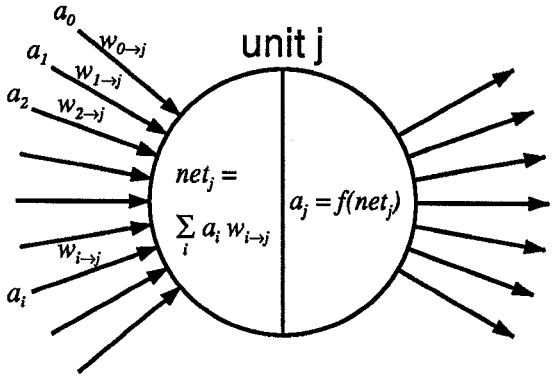

Figure 1.1: A single unit in an artificial neural net

We denote the activity of unit j by a_j, the net input to unit j by net_j, and the weights of the links feeding into unit j by $w_{i\to j}$. The net input to a

unit is calculated in the following way:

$$net_j = \sum_i a_i w_{i \to j} \tag{1.1}$$

where the sum is over all the units i feeding into unit j. The activity of a unit is calculated as:

$$a_j = f_j(net_j) = \frac{1}{1 + exp(-net_j + \theta_j)} \tag{1.2}$$

where f is the activation function, which is, as can be seen, a nonlinear function. The function is sometimes called a squashing function since it takes any real number and squashes it to the interval between 0 and 1. This is illustrated in figure 1.2 which also shows the effect of θ_j as a displacement, so that the function is no longer necessarily anti-symmetrical around zero. By adding a displacement to the net input it is possible to control the degree of activity in a unit that does not receive any input from the units feeding into it. This displacement is usually modeled as an extra link (with weight θ_j) feeding into each unit j from an always active, imaginary unit, the so-called *bias* unit.

A complete discussion of why the activation function is calculated in this way can be found in [Rumelhart, chapter 8]. For the sake of clarity we will omit the displacement θ_j from all following equations.

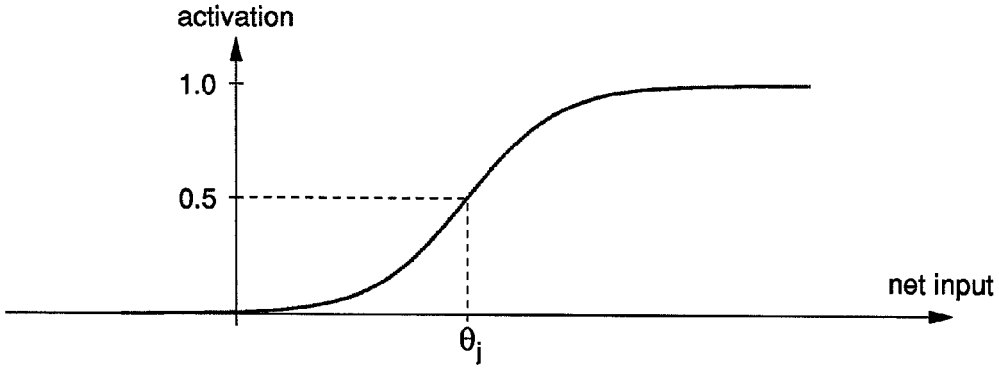

Figure 1.2: The nonlinear threshold function

The dynamic behaviour of a single unit can be implemented as a process in the programming language Occam (See appendix A for a discussion of

OCCAM and transputers). The OCCAM code is given in figure 1.3. The links feeding into a unit are implemented as OCCAM communication links, called *channels*. The weights of the links are stored in the process. The process receives the activity of the units feeding into it over the `input.link` channels and sends the calculated activity over the `output.link` channels to the units that this unit stimulates. The `bias.weight` value is equivalent to the displacement θ_j, and `BIAS.UNIT.ACTIVATION` is equal to 1.

The unit process in figure 1.3 begins with some initialization, primarily a setup of the weights, and the unit then receives in parallel the activities of all units feeding into the unit. After all activities have been received the unit is able to calculate its own activity, which is then sent to all the units it feeds into. In a real net with a number of interconnected units, this propagation of activity will be performed many times. The unit process in figure 1.3, however, makes only one propagation.

```
[NUMBER.OF.INPUT.LINKS]CHAN OF REAL64  input.link:
[NUMBER.OF.OUTPUT.LINKS]CHAN OF REAL64 output.link:

PROC unit()
  [NUMBER.OF.INPUT.LINKS]REAL64 weight, input.activity:
  REAL64 net.input, activity, bias.weight:
  SEQ
    ...  Initialize
    PAR i = 0 FOR NUMBER.OF.INPUT.LINKS
      input.link[i] ? input.activity[i]
    net.input := (-bias.weight) * BIAS.UNIT.ACTIVATION
    SEQ i = 0 FOR NUMBER.OF.INPUT.LINKS
      net.input := net.input + (input.activity[i] * weight[i])

    activity := calculate.activity(net.input)   -- a_j := f(net_j)
    PAR i = 0 FOR NUMBER.OF.OUTPUT.LINKS
      output.link[i] ! activity
:
```

Figure 1.3: A unit as an OCCAM process

1.3 Feed-Forward Nets

Many kinds of artificial neural networks exist, each characterized by the choice of net topology, and the types of activation and learning rules used. In this thesis we will focus on one class of networks only, namely the so-called

layered feed-forward networks [Rumelhart], also sometimes called *multi-layer perceptrons*.

These nets are characterized by the division of units into separate layers, the first layer being an *input* layer followed by a number of *hidden* layers and finally an *output* layer. Every unit in a layer receives input from all units in the previous layer and sends output to all units in the following layer. These are the only existing connections in the net. Even though a feed-forward net in general has many hidden layers, only one is used in most applications. Because this is the case, and the generalization to several hidden layers is trivial, we will only discuss feed-forward nets with exactly one hidden layer. Figure 1.4 is an illustration of the topology of a feed-forward net with one hidden layer.

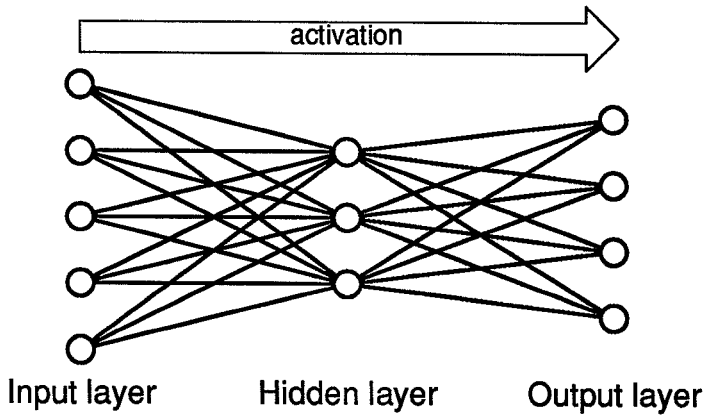

Figure 1.4: Feed-forward network with one hidden layer

It should be noted that, unlike all other units, the units in the input layer (input units) are not computational units. Each of the input units receives only one activity value which is simply spread out to the units that the input unit feeds into, i.e. all the units in the hidden layer (hidden units).

A feed-forward net which has been trained, i.e. the weights have been modified in some way to allow the whole net to respond correctly for a given application, works in the following way: An *input pattern* is fed to the input units in the form of a vector of activities, one activity value for each input unit. The input units simply send these activities to all hidden units. The hidden units calculate their activities as illustrated in figure 1.1 and send these activities to all output units. The output units calculate their activities

in exactly the same way as the hidden units, and the vector of output unit activities is the response put forward by the net.

Let NI, NH, and NO denote the number of input, hidden and, output units, respectively, in a feed-forward net. The propagation of activity is given by the following two equations. For input pattern p the activity of hidden unit j, a^H_{pj}, is calculated as:

$$a^H_{pj} = f(net^H_{pj}) = f(\sum_{i=0}^{NI-1} a^I_{pi} w^H_{i \to j}) \tag{1.3}$$

where $w^H_{i \to j}$ are weights feeding into the hidden layer. Similarly, the activity of output unit k, a^O_{pk}, is calculated as:

$$a^O_{pj} = f(net^O_{pj}) = f(\sum_{j=0}^{NH-1} a^H_{pi} w^O_{i \to j}) \tag{1.4}$$

where $w^O_{i \to j}$ are weights feeding into the output layer.

The input pattern can be a vector of binary values, 0 or 1. This is the case with NETtalk which we will describe in chapter 5. For other training sets the input pattern can be a vector of real values, e.g. values between 0 and 1. In both cases the output unit activities will be real values between 0 and 1. These values can either be used directly as input to various external devices or will have to be interpreted in some application dependent way.

1.4 Training a Feed-Forward Net

In the beginning of a training process the response of the net, when presented with any input pattern, will be a completely random guess. The task of any learning algorithm is to adjust the weights of the net in such a way that the performance of the net reflects the desired function between input and output as closely as possible. The rest of this chapter will be a more detailed description of what "as closely as possible" means and a description of a specific learning algorithm, the *back-propagation* algorithm [Rumelhart].

In this learning scheme (called *supervised learning*) the net very directly is told what is right and what is wrong. This is done with *target patterns*. For every input pattern, there is a matching target pattern for the output units. When the input patterns are presented to the input units and propagated

through the net to produce responses, the net is told the right answers, the target patterns. The net is then supposed to use this information to respond more correctly the next time the same input patterns are presented. To measure how well the net responds to a given pattern, we define the error of pattern p, E_p, in the following way:

$$E_p = \frac{1}{2}\sum_k (t_{pk} - a_{pk}^O)^2 \tag{1.5}$$

where t_{pk} is the target of output unit k for pattern p and a_{pk}^O is the activity of output unit k for pattern p. The sum is over all the output units. The error of all patterns, E, is then defined as:

$$E = \sum_p E_p \tag{1.6}$$

It is now possible to describe in a precise way how the performance of a net is measured. A net performs well when the overall measure of error is small. When E gets smaller the performance gets better. Hence, the task of any learning algorithm is to *minimize* the error function E.

For a given set of input/target patterns, E is only a function of the weights (including the bias weights). Let N be the number of weights in a net, i.e. $N = (NI+1)*NH+(NH+1)*NO$. Consider the $N+1$ dimensional vectorspace given by the weights and the error function E (defined by the weights). The values of the error function will describe a continuous differentiable surface in this vectorspace. This is perhaps best explained by the example in figure 1.5 where the error is only a function of two weights, giving an error function describing an *error surface* in 3-dimensional vectorspace.

There is one set of weights, or maybe several, where E is as small as possible and when such a set of weights is found, the net has learned the task. However, there is no known way to calculate these weights directly, i.e. given the input/target-patterns there is no equation that will produce the right weights. All learning algorithms usually find only a sufficiently close approximation of this set of weights by some iterative search through the N dimensional weight space.

The process of learning begins with an initialization of the weights. They are set to random values, e.g. between -0.5 and 0.5. Then E is calculated given these initial weights and a point on the error surface is defined. By changing the weights one can move around on the error surface. But, the

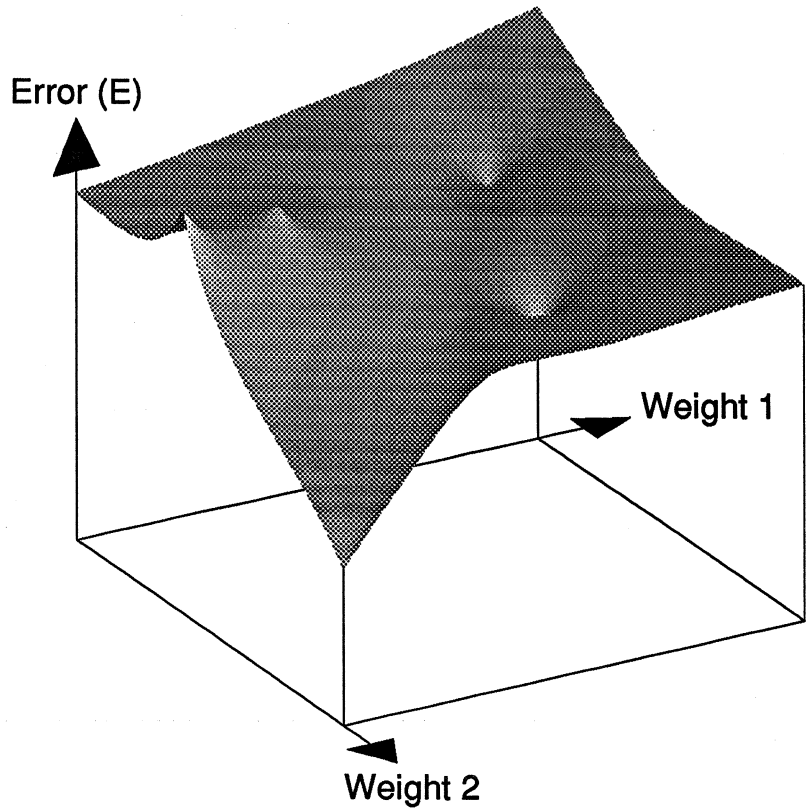

Figure 1.5: Error surface

value of E alone (the height of the error surface in the point given by the weights) gives no hint on how to change the weights. Additional knowledge is required. This additional knowledge is the structure of the surface in a local neighbourhood of the point. A Taylor expansion gives such knowledge; the more terms calculated of the Taylor expansion the more detailed the knowledge of the local structure is. When such knowledge is attained, the algorithm can determine a direction in which to move on the surface and thus find a new set of weights. This process is repeated until a set of weights has been found such that E is sufficiently small.

1.5 Back-Propagation

Moving around on an error surface sounds easy enough, but it was not until 1986 a useful and efficient algorithm dealing with nets with hidden units[2] was found by Rumelhart et.al. [Rumelhart]. The algorithm is a *gradient descent* method. When E is calculated given a set of weights, the gradient in that point is calculated, i.e. only one term in the Taylor expansion is calculated apart from E itself. The weights are changed in proportion to the negative gradient. In this way the method in its simplest form becomes a steepest descent algorithm.

A full discussion of the mathematical background can be found in [Rumelhart] so we will just outline the ideas and give the results.

1.5.1 Calculating Gradients

In the following we will describe how the gradients are computed, since knowledge og these equations is essential in order to be able to parallelize them.

For input pattern p let the change of a weight between arbitrary layers (l and m), $\Delta w_{l\to m}$, be proportional to the negated derivative of E with respect to the weight $w_{l\to m}$:

$$\Delta w_{l\to m} \propto -\frac{\partial E}{\partial w_{l\to m}} = -\sum_p \frac{E_p}{\partial w_{l\to m}} \tag{1.7}$$

where E and E_p are the error functions defined in equations 1.6 and 1.5, respectively.

The application of the back-propagation learning rule involves two phases: During the first phase the input, is presented and propagated forward through the net to compute the output unit activity values. These values are then compared with the targets, resulting in an *error* value $e^O_{pk} = t_{pk} - a^O_{pk}$ for each output unit. This error value is then used in computing a so-called *delta* value:

$$\delta^O_{pk} = f'(net^O_{pk})(t_{pk} - a^O_{pk}) = a^O_{pk}(1 - a^O_{pk})(t_{pk} - a^0_{pk}) \tag{1.8}$$

[2]Algorithms dealing with nets consisting of only an input and an output layer have been known since 1959, but these nets are incapable of learning some complex tasks, e.g. the XOR-problem. See [Minsky] for a discussion.

used in the calculation of the gradient.

The second phase involves a backward pass through the net (analogous to the forward pass) during which the delta values are propagated backwards in the net. The units of the hidden layer calculate their delta values in the following way, δ^H_{pj} being the delta value of hidden unit j:

$$\delta^H_{pj} = f'(net^H_{pj}) \sum_k \delta^O_{pk} w^O_{j \to k} = a^H_{pj}(1 - a^H_{pj}) \sum_k \delta^O_{pk} w^O_{j \to k} \tag{1.9}$$

where δ^O_{pk} is the delta value of output unit k and the sum is over all output units.

Now it, is possible to compute the gradient. The derivative of E_p with respect to a weight between the hidden and output layers, $w^O_{j \to k}$, is calculated as:

$$\frac{\partial E_p}{\partial w^O_{j \to k}} = -\delta^O_{pk} a^H_{pj} \tag{1.10}$$

Similarly, the derivative of E_p with respect to a weight between the input and hidden layers, $w^H_{i \to j}$, is calculated as: $-\delta^H_{pj} a^I_{pi}$.

1.5.2 Calculating Weight Changes

The weight changes can now be calculated. When equation 1.10 is combined with equation 1.7 for the weights between the hidden and output layers we get:

$$\Delta w^O_{j \to k} = -\eta \sum_p \frac{\partial E_p}{\partial w^O_{j \to k}} = \eta \sum_p \delta^O_{pk} a^H_{pj} \tag{1.11}$$

where η is a *learning rate* constant, defining the proportionality factor of equation 1.7.

Normally this rule is extended t,o include a *momentum* term α, such that:

$$\Delta w^O_{j \to k}(n+1) = -\eta \sum_p \delta^O_{pk} a^H_{pj} + \alpha \Delta w^O_{j \to k}(n) \tag{1.12}$$

where n indicates the *learning cycle*, i.e. the number of times the weights have been changed. In this way the weight change depends not only on the most recently calculated gradient but, also on previous changes. This provides a kind of momentum in weight space that effectively filters out high-frequency variations of the error surface in the weight space. This turns out to reduce the learning time.

Equations 1.11 and 1.12 are easily generalized to the weights between the input and hidden layers.

The described method is known as the *true gradient method* [Bourrely] and updating weights in this way is also called *epoch updating*. This is the mathematically correct way of updating weights.

Another method exists, however, known as the *stochastic gradient method* [Bourrely]. In this method the weights are changed after each presentation of a single pattern. The expression *pattern updating* is used. The weights between the hidden and output layers are now changed according to the following equation:

$$\Delta_p w^O_{j\to k} = -\eta \frac{\partial E_p}{\partial w^O_{j\to k}} = \eta \delta^O_{pk} a^H_{pj} \tag{1.13}$$

This equation is normally also extended to include a momentum term such that an equation similar to 1.12 emerges:

$$\Delta_p w^O_{j\to k}(n+1) = -\eta \delta^O_{pk} a^H_{pj} + \alpha \Delta_p w^O_{j\to k}(n) \tag{1.14}$$

If weights are changed according to equation 1.14 the direction of the movement is the gradient direction of the error surface defined by E_p rather than the gradient direction given by E. Obviously, different error functions E_p define different error surfaces. Therefore updating the weights according to equation 1.14 for some p will decrease the value of E_p but not necessarily that of E.

The stochastic gradient method may seem strange because it is E we want to minimize, and indeed the method is in no way mathematically correct. However, empirical studies made from the very beginning by Rumelhart et.al. show that the stochastic gradient method outperforms the true gradient method on most realistic applications.[3]

[3]Problems like the parity problem are learned faster with epoch updating than with

The two methods are extremes. It is, of course, possible simply to update the weights once some specific number of patterns have been presented. If the sizes of these sub-sets of training patterns remain constant throughout all learning cycles, one such sub-set is usually referred to as a *batch* of patterns, and the number of patterns used in performing each weight update is called the *batch size*. When this kind of updating is used we speak of using batch updating of the weights.

The number of patterns used in each weight update may also be a variable number depending on some property of the training patterns. This is the case in the NETtalk application described in chapter 5. Weights are updated each time a number of patterns corresponding to all the letters in one word have been presented.

1.6 An OCCAM Implementation of Back-Propagation

To illustrate how the back-propagation algorithm works, we will extend the simple forward pass unit of figure 1.3 to a full scale unit including the backward pass. Inspired by Welch [Welch], who has worked with a simulation of logical circuits in OCCAM, we will feed the artificial neural net with input patterns and target patterns from an external *environment*. This is illustrated in figure 1.6.

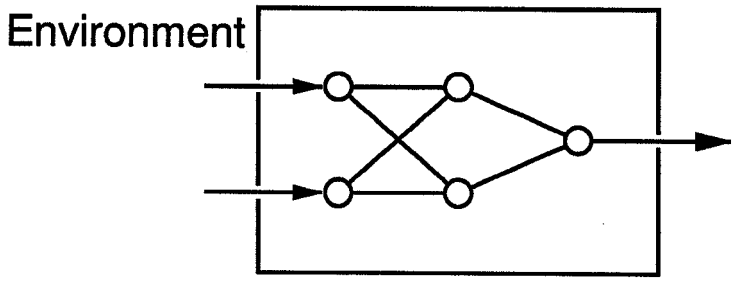

Figure 1.6: An environment controlling an artificial neural network

A net such as the one in figure 1.6 can be trained to solve the XOR-

pattern updating. However, these are generally not very interesting problems to teach an artificial neural net.

problem. There are two input units, two hidden units, and one output unit which result in nine weights (six weights between the five units and three bias weights feeding into the hidden and output units). The XOR-problem consists of only the four patterns given in table 1.1.

Input		Target
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.1: Input/target patterns for the XOR-problem

The XOR-problem may seem very simple and indeed it is no problem at all manually to find values for the nine weights of the net, such that it will respond correctly to all four input patterns. For larger nets, however, there was no general algorithm that could find useful weights, in a reasonable amount of time, before the introduction of the back-propagation algorithm in 1986.

The environment is a process running in parallel with a *simulator* process managing the units (the expression *neural net simulator* is normally used in connection with neural net implementations). The two processes are given in figures 1.7 and 1.8, respectively.

```
PROC Environment([]CHAN OF REAL64 input.link, response.link, target.link)
  ...  Variables
  SEQ
    ...  Initialize
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
        pattern := i REM NUMBER.OF.PATTERNS              -- REM = remainder
        PAR n = 0 FOR INPUT.UNITS
          input.link[n] ! input.pattern[pattern][n]
        PAR n = 0 FOR OUTPUT.UNITS
          response.link[n] ? guess.activity[n]
        PAR n = 0 FOR OUTPUT.UNITS
          target.link[n] ! target.pattern[pattern][n]
:
```

Figure 1.7: The environment process

As can be seen in figure 1.7, the learning phase runs for a fixed number of steps. Each time a new pair of input/target patterns is chosen. The

```
PROC Simulator([]CHAN OF REAL64 input.link, response.link, target.link)
  ...  Variables
  ...  FUNCTION calculate.activity(net.input)
  ...  PROC input.unit(number)     (figure 1.9)
  ...  PROC hidden.unit(number)    (figure 1.10)
  ...  PROC output.unit(number)    (figure 1.11)

  PAR
    PAR n = 0 FOR INPUT.UNITS
      input.unit(n)
    PAR n = 0 FOR HIDDEN.UNITS
      hidden.unit(n)
    PAR n = 0 FOR OUTPUT.UNITS
      output.unit(n)
:
```

Figure 1.8: The simulator process

input pattern is sent to the input units (via the `input.link` channels). The net's response is collected from the output units (via the `response.link` channels). Finally, the correct target pattern is sent back to the output units (via the `target.link` channels).

The feed-forward unit process of figure 1.3 can be extended to include the backward pass. Since there is a difference in the way delta values are calculated depending on the type of the unit, we have programmed three different unit processes – an input unit process, a hidden unit process, and an output unit process. These are given in figures 1.9, 1.10, and 1.11, respectively. The units obey the pattern updating scheme.

The links for communicating internally between the unit processes are placed in two-dimensional arrays called `hidden.link` and `output.link`. The hidden links are the links feeding into the hidden units and the output links feed into the output units.

```
PROC input.unit(VAL INT number)
  REAL64 activity:
  SEQ i = 0 FOR NUMBER.OF.ITERATIONS
    SEQ
      input.link[number] ? activity
      PAR j = 0 FOR HIDDEN.UNITS
        hidden.link[number][j] ! activity
:
```

Figure 1.9: An input unit

```
PROC hidden.unit(VAL INT number)
  ...  Variables
  SEQ
    ...  Initialize
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
        ...  Propagate activity          (figure 1.12)
        ...  Calculate weight changes    (figure 1.13)
        ...  Change weights
:
```

Figure 1.10: A hidden unit

```
PROC output.unit(VAL INT number)
  ...  Variables
  SEQ
    ...  Initialize
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
        ...  Calculate activity
        response.link[number] ! activity
        target.link[number] ? target
        ...  Calculate weight changes
        ...  Change weights
:
```

Figure 1.11: An output unit

```
{{{  Propagate activity
PAR j = 0 FOR INPUT.UNITS
  hidden.link[j][number] ? input.activity[j]
net.input := (-bias.weight) * BIAS.UNIT.ACTIVATION
SEQ j = 0 FOR INPUT.UNITS
  net.input := net.input + (weight[j] * input.activity[j])
activity := calculate.activity(net.input)
PAR j = 0 FOR OUTPUT.UNITS
  output.link[number][j] ! activity
}}}
```

Figure 1.12: Hidden unit activity propagation

Figures 1.12 and 1.13 are fold expansions of the corresponding folds in figure 1.10. Note that the displacement of the activation function as given in equation 1.2 is changed just as any other weight. The learning rate and momentum terms are normally set to 0.2 and 0.9 respectively, see [Rumelhart].

```
{{{  Calculate weight changes
PAR j = 0 FOR OUTPUT.UNITS
  output.link[number][j] ? weighted.delta[j]
total.delta := 0.0(REAL64)
SEQ j = 0 FOR OUTPUT.UNITS
  total.delta := total.delta + weighted.delta[j]
delta := (total.delta * activity) * (1.0(REAL64) - activity)
SEQ j = 0 FOR INPUT.UNITS
  weight.change[j] := (momentum * weight.change[j]) +
                      (learning.rate * (delta * input.activity[j]))
bias.change := (momentum * bias.change) +
               (learning.rate * (delta * BIAS.UNIT.ACTIVATION))
}}}
```

Figure 1.13: Hidden unit weight change calculation

Like the environment process the unit processes now run for a fixed number of iterations. The complete program can be found in appendix B.1.

Even though the overhead involved with process managing on the transputer is very low, the program as sketched above with individual processes for each unit in an artificial neural net is not very efficient. We have also programmed a standard implementation of the back-propagation algorithm, i.e. a sequential, non-process oriented program, which can be found in appendix B.2.

We have run both versions on the XOR-problem and nets of larger sizes. In table 1.2 are the results.

Net size	Standard implementation	Process oriented implementation
XOR-problem	0.24 sec	0.33 sec
10-10-10	3.32 sec	9.22 sec
20-20-20	12.70 sec	38.44 sec

Table 1.2: Comparison of standard and process oriented versions

The execution times in table 1.2 are for 1000 iterations and the pattern updating scheme has been used. The net 10-10-10 is a net with 10 input units, 10 hidden units, and 10 output units. Likewise for the 20-20-20 net. The XOR-problem is a "real" problem and the net actually learns the XOR-function. This is not the case with the other two nets. They are simply nets of convenient sizes with pseudo learning tasks.

The process oriented version is only 40% slower than the standard im-

plementation of the XOR-problem (a 2-2-1 net). However, it is three times slower for the larger nets. This is easily explained, because the calculations in the units are identical for the two versions, and the overhead for the process oriented version is due to communication over the links. In the 2-2-1 net there are 5 units and 6 links and in the 10-10-10 net there are 30 units and 200 links. The links/units ratio is much larger in nets with more units and thus the time used to perform the link communications becomes essential.

Although the process oriented version is parallel by nature, we will not use this version or extend it when we develop versions for running on several transputers, due to its slowness.

1.6.1 Analysis of the Sequential Program

The back-propagation algorithm uses floating point operations to a very high degree. We will now analyze how well our implementation of the algorithm makes use of the transputer's floating point capabilities. Table A. 1 in appendix A (page 130) gives the speed of the four basic floating point operations. We will use these in the following.

In the forward pass the calculations, as given by equations 1.3 and 1.4, use 1 addition and 1 multiplication per weight in both layers. In addition, 33.5 μsec is used per unit in the hidden and output layers to calculate the activation function. This high number mainly stems from the calculation of the exponentiation function.

In the backward pass the calculations of delta values for output and hidden units, as given by equations 1.8 and 1.9 use 2 subtractions and 2 multiplications per output unit, 1 subtraction and 2 multiplications per hidden unit, and finally 1 addition and 1 multiplication per weight feeding into the output layer.

When calculating the weight changes, as given by equation 1.14, the algorithm uses 1 addition and 3 multiplications per weight in both layers. The actual changing of the weights requires just 1 addition per weight in both layers. All these numbers are summarized in table 1.3. The last column gives the total times used per unit and weight.

To see how well we utilize the transputer's floating point capabilities, we examine the simulation of the 20-20-20 net. In this net there are 20 units in each layer. There are 420 weights between the input and hidden layer (400 weights between the units of two layers and 20 bias weights feeding into the hidden layer units). Likewise for the weights between the hidden and output

Operation count	+/−	∗	Total time
Per hidden unit	1	2	35.1 μsec
Per output unit	2	2	35.4 μsec
Per weight between input and hidden units	3	4	3.5 μsec
Per weight between hidden and output units	4	5	4.4 μsec

Table 1.3: Floating point operations used in pattern updating

layers, giving a total 840 weights in the net.

This results in a total of 3000 additions and 3860 multiplications. With the extra 33.5 μsec used per unit in the hidden and output layers, a total of 0.0047 seconds is the theoretical lower bound on the calculation time for one forward and one backward pass. The execution times of table 1.2 are for one thousand iterations. With an execution time of 12.7 seconds our implementation utilizes 37% of the available floating point operations capacity. This is not impressive but still a good utilization. Additionally, the implementation consists of more than floating point operations. There are substantial index calculations, all weights are copied for each iteration, and so forth.

For comparison, Christiansen and Tolbøl [Christiansen] have implemented an algorithm for calculating the Mandelbrot set. Like neural network simulators this algorithm uses floating point operations to a very high degree. Christiansen and Tolbøl were able to utilize 46% of the transputers' floating point capabilities. By optimizing critical parts of the algorithm, i.e. implementing the parts directly in machine code, they were able to increase the performance by 26% (from 46% to 58%).

For comparison of execution times, Petrowski et.al. [Petrowski] give the execution time of their sequential back-propagation implementation on a transputer. The transputers they are using are T800-20 (20 MHz) whereas we are using T800-30 (30 MHz). When executed on nets of varying sizes, our implementation is between 1.86 and 2.07 times faster. If we assume that the T800-30 processor is 50% faster than the T800-20 then our implementation is still faster (between 1.24 and 1.38 times).

Chapter 2

Parallelizing Algorithms

2.1 Parallelization Strategies

Whenever one is trying to transform a sequential algorithm into a parallel one, there are at least two possible strategies worth considering. In the neural net context these two strategies are usually referred to as *data partitioning* and *net partitioning*, respectively.

2.1.1 Data Partitioning

If the same algorithm is to be applied a large number of times on different sets of data, then it is often very efficient to run these tasks concurrently on different processors, provided that the tasks are truly independent, i.e. the execution of one task does not depend on the results of the other tasks. This parallelization strategy is sometimes referred to as *job-level* parallelism [Forrest] or *data partitioning* [Pomerleau1].

If the weights in a neural network are only updated after the presentation of several patterns, each of those pattern presentations are independent tasks that may be carried out concurrently. Therefore if epoch or batch updating of the weights are used during the training run of some neural network, then the data partitioning approach is very easily applied: The training data are simply distributed evenly among the available processors, all of which simulate the entire network but on different sub-sets of training data. Once in a while the results of presenting the various groups of patterns are combined and a weight update is performed.

When applied to neural networks the data partitioning strategy is also

referred to as *training parallelism* [Millán] for obvious reasons.

It is worth noting at this point, that data partitioning is not possible if the weights are updated after each pattern has been presented. When pattern updating is used the state of the network is changed as a result of each pattern presentation. Therefore the result of presenting one pattern depends on the results of all patterns presented earlier, which means that the tasks of presenting individual patterns no longer can be considered independent.

We are going to describe and analyze the properties of a number of parallelizations that exploit the data partitioning strategy in chapter 3.

2.1.2 Net Partitioning

Another way of employing parallelism is to use the so-called *geometric* [Forrest] or *spatial* [Millán] parallelism. In this approach it is the execution of the algorithm on *one* set of data which is parallelized. This is done by distributing the data amongst the processors in such a way, that all data required by a processor are stored in that processor or is easily accessible from one of the neighbouring processors when needed.

When applied to neural networks this strategy is often called *net partitioning* [Pomerleau1], since the processing of one pattern can be parallelized by dividing up the network, letting each processor handle a small part of the net.

A discussion of different ways of cutting up the network as well as a detailed description of (the construction of) an implementation of the net partitioning strategy for parallelizing neural networks can be found in chapter 4.

2.2 Analyzing Parallel Algorithms

We will now introduce some concepts that will be useful in analyzing the performance of parallel neural network algorithms. The primary concern is how well the resources of the extra processors are exploited. The degree of exploitation is called the *efficiency*. When analyzing a parallel algorithm we are interested in varying a number of parameters in order to find out how the efficiency of the algorithm is influenced by those parameters. Examples of such parameters are neural network specific parameters like the number of units in each layer, the number of weights, the frequency with which weights

are updated, and so on. There are also parameters which are related to the parallelization itself, most notably the number of processors used in executing the algorithm, and how those processors are configured, i.e. the pattern of processor inter-connectivity.

In order to give the formal definition of efficiency it is necessary to know a bound on how much faster we can expect a parallel algorithm with P processors to perform. We will therefore introduce another often used concept, namely that of *speed-up* [Fox1] before giving the formal definition of efficiency. Since the primary goal of any parallelization is to reduce the running time, a natural way of measuring the performance of a parallel algorithm is to directly compare its execution time on some specific problem with that of the corresponding sequential algorithm, so as to determine how much faster the parallel algorithm is.

More formally, the speed-up of a parallel algorithm on a specific problem is defined to be the ratio of the execution time T_{seq} of the sequential algorithm to the execution time T_{par} of the parallel algorithm when both algorithms are applied to that problem:

$$S(P) =_{def} \frac{T_{seq}}{T_{par}(P)} \tag{2.1}$$

where P is the number of processors used in executing the parallel algorithm. In general, the sequential algorithm used should be the fastest known. However, in order to be able to use this measure of speed-up in evaluating whether we have been successful in parallelizing some specific algorithm, we will put a number of further restrictions on how T_{seq} should be obtained. Thus, we require that the sequential algorithm should be implemented in the same programming language as the parallel version, and the processor running this sequential algorithm should be identical to the processors used in executing the parallel algorithm. Furthermore, the sequential and parallel algorithms must be run on exactly the same data, and with identical neural net specific parameters, including the frequency of weight updates.

Any parallelization of the back-propagation algorithm must contain all the computations found in the sequential algorithm. Therefore, if the only cause for the speed-up of a parallel neural net algorithm is the fact, that calculations can now be performed concurrently, then obviously it follows that the speed-up $S(P)$ of some parallel algorithm with P processors is bounded by the value of P. With P times as many computational resources the best we can hope to achieve is a reduction of the execution time by a factor of P.

However, it should be noted that the execution time of a parallel algorithm may sometimes be reduced by other circumstances related to hardware specific properties of the processors used. One such example is the small and fast on-chip memory found on each transputer (see appendix A.1). If the on-chip memory is used to store part of the units or weights of the neural network and a net partitioning parallelization approach is used, then we might observe a speed-up of more than P with P processors due to the fact, that as more processors are used a still larger fraction of the neural net may be stored in the faster on-chip memory.

To avoid any difficulties with phenomena like the above (and since the effect of storing a fraction of the net or part of the program in on-chip memory is relatively small) we have decided to make no use of the on-chip memory in the transputers, i.e. we have explicitly filled the on-chip memory with "garbage", so that no part of the transputer system might try and use this memory "behind our back". Only exception is the runs made on the NETtalk data set (see chapter 5), since those runs are not intended for analysis of the parallel algorithm, but merely for comparison with the results obtained on other parallel architectures. Whenever nothing specific is mentioned, on-chip memory is not used.

With the above definitions, we are now able to express efficiency as the ratio of observed to optimal speed-up. If we assume that speed-up is only due to the effects of concurrent computations, i.e. that $S(P)$ is bounded by P, we can give the following formal definition of efficiency:

$$E(P) =_{def} \frac{S(P)}{P} = \frac{T_{seq}}{T_{par}(P)P} \tag{2.2}$$

As can be seen the value of $E(P)$ is bounded by 0 and 1 (provided that our assumption holds).

2.3 Main Objectives of a Parallelization

The most important reason for parallelizing an algorithm is usually the desire to attain a reduction in the execution time of that algorithm. Such a reduction in the time required to let the algorithm process some set of data is desirable since it will not only allow more tasks of the same size to be undertaken, it will also make practical the handling of computationally larger tasks [Fox1].

Also, larger tasks with respect to memory requirements (i.e. problems with larger data sets) can be managed if the parallelization allows the data on which the algorithm works to be distributed among the available processors, thereby reducing the memory demands of individual processors. Therefore, the total memory requirement of a parallel algorithm should preferably be no larger than that of the sequential algorithm.

If an efficient parallelization of an algorithm can be devised it is an easy and cheap way of speeding up the execution of the algorithm. Provided that the efficiency of the parallel algorithm is preserved even when large numbers of processors are used, a system of parallel processors will often be able to out-perform one single powerful computer. Moreover, a parallel system is usually easily extended, so that extra speed can be attained by adding a few extra processors to the system.

However, parallelizing algorithms is generally not a trivial task. A number of circumstances have to be considered, including the properties of the physical hardware available: If the algorithm is to run on a shared-memory parallel computer, it is necessary to take into consideration whether concurrent read and write operations are allowed, and if so, at what cost. If, on the other hand, the available computer is a distributed-memory multi-processor machine (as the transputer system) in which each processor has its own memory and no direct access to the memories of other processors, then it is important to know how fast the inter-processor communication is, compared to the computational capabilities of each individual processor. Especially so, if (as is the case with the transputers) communication and calculation can be performed concurrently, since this will allow communication to take place at little cost as long as the time required is smaller than the computation time (see appendix A.4).

Also, there will most likely be restrictions as to how the processors may be configured, i.e. how they may be wired together. An easily extendable processor configuration is preferable since this will allow extra processors to be put to use as soon as they become available. Also, if the number of processors can be chosen completely at will, it is often easier to distribute the neural net problem in even shares to all processors. Therefore, in general, architectures like a hypercube topology should be avoided unless there are some other large advantages to be gained from using such a processor configuration. This is also the case with topologies like the two-dimensional torus and mesh where the number of processors must be a multiple of two integers, preferably two identical integers so that the two dimensions in the torus or

mesh are of equal size.

The most dynamic of processor topologies is the ring configuration in which the number of processors can be chosen arbitrarily. Also easily extendable is the topology in which processors are configured as a binary tree, although such a tree cannot always be completely balanced.

In addition to the issue of extendability another issue worth considering when choosing processor topology is the cost of non-local communication. This issue turns out to be less important, since it is possible for us to construct all algorithms (expect the algorithm discussed in section 3.3) so that, when needed, data are always available in neighbouring processors without requiring any extra communications.

2.4 Sources for Inefficiency in a Parallel Algorithm

There are a number of reasons why a parallel algorithm may not utilize the available computational resources as efficiently as the corresponding sequential algorithm. When a lack of efficiency is observed in some specific algorithm it will most likely be the result of several of the causes for inefficiency described below. We will discuss what causes apply to what algorithms in the relevant sections on these algorithms.

2.4.1 Software Overhead

It may be necessary to introduce additional or more complex index calculations in each processor in order to handle data originating from various other processors. Or the sequence of calculations may have to be altered for some reason (see section 2.4.3), so that some temporary results perhaps no longer are available when they are needed again and therefore have to be re-calculated. Any such extra work will reduce the efficiency of the algorithm.

However, if software overhead constitutes a constant fraction of the work performed in each processor regardless of how many processors are used, then the total amount of work pertaining to software overhead is independent of the number of processors. Since such work is necessarily performed in parallel the efficiency of the parallel algorithm is always reduced by the same constant factor, irrespective of the number of processors used in executing

the algorithm:

$$E(P) = \frac{T_{seq}}{T_{par}(P)P} = \frac{T_{seq}}{\frac{T_{seq}+T_{soft}}{P}P} = \frac{T_{seq}}{T_{seq}+T_{soft}} \tag{2.3}$$

In the above equation we have assumed that software overhead independent of the number of processors is the only cause for inefficiency in the parallel algorithm. T_{soft} is the time required for one processor to perform the work associated with the total amount of software overhead.

Since efficiency is reduced by a *constant* factor, this kind of software overhead cannot put a limit to the speed-up that can be achieved on some given neural net problem. The presence of such software overhead merely results in a fixed poorer utilization of each individual processor involved in running a neural net simulation.

If, on the other hand, the software overhead in each processor is not reduced as much as the number of processors is increased then software overhead will constitute a growing fraction of the work performed, both in each individual processor and in the system as a whole. In this case, efficiency will deteriorate as the number of processors is increased.

2.4.2 Load Balancing

A system of concurrently working processors has not finished until *all* processors have finished. Therefore it is important to ensure that the workload is distributed evenly among the processors, so that each processor performs the same amount of work. Moreover, the workload should be balanced evenly among the processors *at all times* during the execution of the algorithm, since otherwise the processors may simply alternate between working and waiting in such a way, that even though all processors have handled equal shares of the total workload they have not done so fully in parallel.

Load balancing problems may or may not become more pronounced as the number of processors grows, depending on the processor configuration used and the nature of the neural net problem.

2.4.3 Communication Overhead

Any time used for communication will reduce the efficiency of the algorithm, since this is extra work as compared to the sequential algorithm. This means,

that one should try to minimize the amount of inter-processor communication, at least if such communication cannot take place concurrently with some of the computational work.

On the transputers, though, communication and computation may generally be interleaved so that the cost of communicating may become nearly insignificant, provided that no computations have to wait for the communications to finish. Therefore an important consideration in the construction of parallel algorithms for use on transputers is the rearrangement of the sequence of necessary computations in order to be able to achieve the highest possible degree of concurrency in the performance of communications and calculations. Sometimes, however, there may not be any computation left that does not depend on the data being communicated at this point. Also, it is worth noticing that although the cost of communication can be reduced significantly by performing it concurrently with calculation, the cost never becomes completely negligible. See appendix A.4 for a full discussion of this.

In several of our parallelizations the amount of data communicated by each processor does not depend on how many other processors there are. That is, the time used for communication in each processor is not reduced when the number of processors is increased. Since each processor's fraction of a given neural net problem becomes smaller and smaller as more processors are used in simulating the neural net, this means that a growing fraction of the running time is spent on communication, leading to a steady decrease in efficiency.

Let us for the sake of the argument assume that for some parallel algorithm the only cause for less than optimal efficiency is the communication overhead. That is, everything else but communication is perfectly parallelized. Furthermore, let us assume that the time used in each processor for communicating with other processors depends only on the size of the neural net problem, i.e. that the time spent on communication in each individual processor is independent of the total number of processors. We can now express the speed-up of such an algorithm in the following way:

$$S(P) = \frac{T_{seq}}{T_{par}(P)} = \frac{T_{seq}}{\frac{T_{seq}}{P} + T_{comm}} \tag{2.4}$$

Note, that if T_{comm}, the time spent in each processor on communication, is zero then speed-up is optimal. On the other hand, if T_{comm} is different from zero communication overhead will effectively have put an upper limit,

T_{seq}/T_{comm}, to the speed-up that can be achieved using this algorithm on some specific neural net problem, no matter how many processors are used.

In general, the considerations about the effects of software overhead also apply to the issue of communication overhead. If the time used for communication in each processor is not reduced as much as the number of processors is increased then this communication overhead will cause the efficiency to deteriorate as more processors are used.

2.4.4 Inherently Sequential Parts of the Algorithm

There may be inherently sequential parts of the algorithm, i.e. parts of the algorithm that simply cannot be parallelized, e.g. because the execution of each step in the algorithm depends on the completion of the previous step. In a neural net context such inherently sequential parts may often be found in the initial distribution of weights or training patterns, as well as in the final collection of partial results from individual processors. Another example may be the calculation of a global scalar product (as found in the conjugate gradient [Johansson] learning algorithms for neural networks). It is worth noticing that an inherently sequential part of the algorithm may sometimes be parallelized in the sense that all processors carry out the same computations, each processor performing the sequential part on its own, computing the result all by itself. This does not, however, constitute a true parallelization, since the running time required for obtaining the result is not reduced as compared to the sequential algorithm. Though it may be a more elegant (and efficient) solution than having one single processor perform the computations since this would require a broadcast of the result to all other processors afterwards.

If an algorithm contains inherently sequential parts, the time required for executing these parts will form a limit to the speedup that can be attained, no matter how many processors are used. This is usually known as *Amdahl's law* [Fox1]. It states that if some inherently sequential part of the algorithm takes fraction $1/\alpha$ of the running time when the algorithm is executed on one processor, then it will not be possible to obtain a speed-up larger than α. As more and more processors are used the time necessary for executing the sequential part will take up a larger and larger part of the running time, thereby reducing efficiency.

2.4.5 Problem Specific Limitations

It is important to realize that as long as only one parallelizing strategy is used, there is always a theoretical limit to the speed-up that can be achieved on a problem of fixed size.

Any parallelization consists of dividing up the problem into a number of sub-problems, that can be distributed among the available processors. Different parallelization strategies split the original problem in different ways, along different dimensions, so to speak. For each dimension, each parallelization strategy, there is a limit to the number of sub-problems into which the original problem can be decomposed. This number of sub-problems forms an upper bound on the number of processors that can be used profitably in the parallel algorithm, since each processor should handle at least one single sub-problem.

As an example, consider a simple implementation of the data partitioning strategy for parallelizing neural networks. In the context of such an algorithm, the presentation of a pattern in the current batch of training patterns represents one sub-problem which can be solved by a single processor. However, in this case the original problem cannot be divided into more such sub-problems than there are patterns in the batch, since each processor must handle at least one pattern (or it will do nothing more than simply disturb the processors that do have a part of the batch). In other words, using only the data partitioning approach we can never expect a speed-up larger than the batch size used.

Likewise, we may experience in an implementation of the net partitioning approach that we are unable to apply more processors than there are units in the neural net, this way limiting the speed-up that can be achieved to the number of units in the net.

It should be noted that the above limitations are mostly theoretical, since usually speed-up is bounded by the effects of a number of other reasons for inefficiency, most notably software and communication overhead.

2.5 Neural Net Specific Considerations

Usually when parallelizing algorithms it goes without saying that the only difference between the sequential and parallel algorithms should be the speed with which they are executed. In other words, the two algorithms should be

functionally identical, i.e. they should produce the same output.

In the neural net context this means that the quality of learning in terms of generalizing ability and learning speed (measured in number of pattern presentations) should be preserved by the parallelization of the algorithm. However, strange as it may sound, this is actually not always the case for parallelized neural net learning algorithms.

The problem occurs because of the varying frequency with which the weights in the neural net are updated. The quality of learning is heavily influenced by how many patterns are presented to the net between weight updates (i.e. by the size of the batch of training patterns) as we shall see in chapter 5. Of course the sequential back-propagation algorithm may use whatever frequency of weight updating is suitable for the specific neural net task undertaken. However, in parallelizations of back-propagation (especially of the data partitioning kind) the weights are often not updated with the frequency most suitable to the given neural net problem. Instead, the number of pattern presentations between weight updates is chosen so as to suit the parallelization. As a result of this, the size of the batch usually varies with the number of processors used in executing the parallel algorithm.

Neural network learning algorithms are parallelized because of the desire to achieve an increase in the speed (measured in absolute time) with which the network can be trained to perform a given task to a certain degree of perfection. Therefore, as a consequence of the described effect of varying the batch size, the correct way of comparing different parallel algorithms is *not* by measuring the number of patterns that each algorithm can present to the net per second, because this does not take into account any differences in the effect on learning of each pattern presentation. Ideally, parallel algorithms should be compared by measuring how fast each algorithm could train a given neural net to perform a certain task to some degree of perfection. However, the effect of batch size on the quality of learning varies in different neural net applications. Consequently, if the two parallel algorithms to be compared use different batch sizes our evaluation of them will depend on what neural net problem we use as a benchmark.

The conclusion to the above considerations is, that in order to really perform a fair comparison of two parallel algorithms (whether running on the same system of processors or not) they should be applied to the same neural net problem, and all parameters should be identical, most notably the size of the batch.

2.6 Experiments for Performance Analysis

In general, we wish to enquire about the effect of all kinds of parameters on the efficiency of the parallel algorithm, be it neural net specific parameters like the size of the batch, or hardware specific parameters like the structure of the processor configuration or the number of processors used in executing the algorithm.

As we noted in section 2.3 there are two main reasons for parallelizing an algorithm. One is to reduce the time required for applying the algorithm to some specific problem. The other is to allow an increase in the size of the problems that can be undertaken.

2.6.1 Fixed Problem Size

For a number of fixed problems we will show the effects of increasing the number of processors used in executing the parallel algorithm on the problem concerned. If the fraction of time wasted in each processor is proportional to the number of sub-problems handled by that processor, then speed-up graphs will be straight lines, i.e. speed-up will be linear in the number of processors (as long as there are fewer processors than sub-problems), and the efficiency graph will be a horizontal, straight line, indicating that efficiency is independent of the number of processors used.

If, on the other hand, the fraction of time wasted in each processor grows with the number of processors, then efficiency deteriorates as more processors are used. This will make the speed-up graphs curve, making them look similar to the graphs in figure 3.5 on page 47.

Whenever possible we will try to interpret the graphs in order to find out which of the circumstances mentioned in section 2.4 plays the most important role in reducing the efficiency of the algorithm in question.

2.6.2 Variable Problem Size

We would also like to examine the effects of scaling the size of the neural net problem with the number of processors in order to determine how much larger a problem can be handled (with same degree of efficiency) when more processors are used. Ideally, it should be possible to use twice as many processors when the size of the problem is doubled. However, as we shall see, it is difficult to measure the size of a neural net problem.

More precisely, the *size* of a problem is the amount of computational work associated with solving the problem (using the sequential algorithm). When speaking of neural net problems, *solving the problem* means training the net until a certain degree of perfection is reached.

For this reason it is usually very difficult to compare the sizes of two neural net problems, since the inclusion of extra units in the net or an increase in the size of the batch may or may not change the number of learning cycles needed to attain some pre-set level of perfection, depending on what learning task the net is trained to perform.

For some specific learning task, we cannot simply assume that a net containing twice as many hidden units represents a problem (nearly) twice as large[1] since both too few and too many hidden units may increase the number of learning cycles required to obtain a given performance.

Similarly, increasing the batch size may lead to a poorer performance of the net with respect to the number of learning cycles necessary to obtain a given level of perfection (see chapter 5 about the NETtalk learning task). That is, doubling the batch size may increase the computational size of the problem by much more than a factor of two. Or it may, in some other neural net application, actually lower the number of learning cycles required to produce a solution, perhaps thereby even reducing the size of the problem.

Because of the difficulties described above we will refrain from trying to measure how efficiency varies when *problem* size is scaled with the number of processors. Instead, we will simply examine how efficiency is influenced by varying parameters like net size and batch size.

2.6.2.1 Variable Number of Sub-Problems

Instead of worrying about how efficiency will vary when the *size* of the problem is scaled with the number of processors, we will examine to what degree efficiency is preserved when the number of *sub-problems* is scaled with the number of processors. In other words, we will examine if more processors can be put to use without reducing efficiency, as long as each processor handles the same number of sub-problems.

As we have stated, even if some new problem can be split into twice as many sub-problems, the size of this problem is not necessarily twice as

[1]When the number of hidden units is doubled, so is the number of weights in the net. However, since the number of output units is unchanged, the amount of computation in each learning cycle will only *nearly* have doubled.

large. In a net partitioning parallelization the number of sub-problems is doubled by doubling the number of units (in each layer of the net). We cannot know the effect on the size of the problem, since this depends on the neural net application in question. But what we *can* and *do* know is that doubling the number of units means increasing the number of weights four-fold, causing the presentation of one pattern to be nearly found times as demanding computationally (in large nets).

In a data partitioning parallelization a doubling of the number of sub-problems is obtained by doubling the size of the batch. Again, the influence on problem size depends on the actual neural net application. We may state, however, that since the number of pattern presentations between weight updates is doubled, the amount of computational work associated with one single learning cycle will have doubled also (or rather, nearly have doubled, since the work associated with the actual updating of the weights is unchanged).

In a sense, therefore, some new problem containing a larger number of sub-problems is not so much a larger problem, as it is simply a *different* problem with different properties – properties more suitable to the algorithm in question.

For several different fixed numbers of sub-problems per processors we will show the effects of increasing the number of processors. As in the experiments with fixed problem size, we will try to interpret the resulting speed-up and efficiency graphs. If the efficiency graphs are not straight, horizontal lines then we may conclude, that the overhead found in each processor cannot be independent of how many processors are used in executing the algorithm.

Whenever possible we will try to interpret the graphs in order to find out which of the circumstances mentioned in section 2.4 plays the most important role in reducing the efficiency of the algorithm in question.

2.6.2.2 Fixed Number of Sub-Problems

We are also interested in examining the effects of varying neural net specific parameters that do not influence the number of sub-problems, since knowledge of such effects are necessary if a neural net researcher is to choose the most suitable parallelization for his or her application. Therefore we will also examine what happens in the data partitioning parallelizations when the size of the net is changed, and how efficiency is influenced by varying the batch size in the net partitioning parallelization.

2.6.3 Experimental Conditions

The distribution of patterns were excluded from all time measurements, since we are interested in knowing only the raw time of actually training a neural net. Also, the time needed for distributing patterns depends on what neural net application is used. Furthermore, the fraction of total running time used in distributing patterns depends on the number of learning cycles in the following training session of the net.

In order to be able to vary the number of units and weights in the net freely, we have chosen to run all algorithms on pseudo problems in which all training patterns were generated by a random generator.

Because of the quite large number of experiments we have performed and the literally thousands of results obtained from these experiments we have chosen to present all results as graphs only.

Chapter 3

Back-Propagation Using Data Partitioning

As mentioned in chapter 2 an obvious and easy way of parallelizing an artificial neural network using epoch or batch updating of the weights is the so-called *data partitioning* strategy in which the training data are distributed evenly among the processors. All processors simulate the entire network but on different sub-sets of the training data. During each learning cycle each processor presents the patterns in its own share of the current batch.[1] The gradients calculated this way in the individual processors are called *component gradients*, since each of them is the result of presenting only some part of the batch to the network. Once all patterns in the batch have been presented by the various processors, the resulting component gradients are combined (summed) into one global gradient, which is then used to calculate how much each weight should be changed. Following that, the weights are updated and broadcast to all processors as the new set of weights that should be used in the subsequent presentation of patterns during the next learning cycle.

This is the simplest form of the data partitioning approach to parallelizing artificial neural networks. Implementations of this form will be discussed in the following section 3.1. In section 3.2 we are going to look at a variation of the data partitioning approach in which each processor does not hold a complete copy of the entire network. Finally, in section 3.3 we are going to

[1] In this section we will not discriminate between epoch and batch updating since, with respect, to the parallelization of an algorithm, epoch updating may simply be viewed as a special case of batch updating in which the batch is always the complete set, of training patterns.

describe a different way of parallelizing that still depends on a batch of input patterns being presented concurrently.

3.1 A Simple Implementation of the Data Partitioning Strategy

The actual process of implementing the simple form of the data partitioning strategy is quite straight-forward once the non-process oriented sequential back-propagation algorithm has been developed (section 1.6). All we have to do is put together a lot of processors, each of which should run the sequential algorithm on its own, independently from the others. With P processors we therefore have P identical copies of the entire network. During each learning cycle, each processor q presents to the network the patterns in its part B_q of the current batch B of training patterns. As a result the following component gradient $\overline{g}^{<q>}$ is calculated in processor q:

$$\overline{g}^{<q>} = \sum_{p \in B_q} \frac{\partial E_p}{\partial \overline{w}} \tag{3.1}$$

where m denotes the set of all weights.

Using equations 1.11 and 1.12 in chapter 1 the weight change $\Delta_B \overline{w}$ with respect to the patterns in batch B can now be expressed as a sum of component gradients $\overline{g}^{<q>}$:

$$\Delta_B \overline{w}(n+1) = -\eta \sum_{p \in B} \frac{\partial E_p}{\partial \overline{w}} + \alpha \Delta_B \overline{w}(n) = -\eta \sum_{q=0}^{P-1} \overline{g}^{<q>} + \alpha \Delta_B \overline{w}(n) \tag{3.2}$$

Therefore, only one modification is necessary as compared to the sequential algorithm: A scheme for performing the collection and summation of all component gradients $\overline{g}^{<q>}$ and the distribution of updated weights (a broadcast) in as effective a way as possible.

3.1.1 Communication Schemes

Different configurations of processors require different communication schemes, depending on what patterns of connectivity are allowed.

Using a ring configuration of P processors it is quite easy to implement the summation of component gradients in such a way that each processor

after $P-1$ steps will hold the total gradient. Initially each processor simply communicates its own component gradient to its predecessor in the ring (figure 3.1). Then all processors add the component gradient just received to a temporary sum of component gradients. At the same time the component gradient is sent along to the predecessor concurrently with the receipt of a new component gradient.

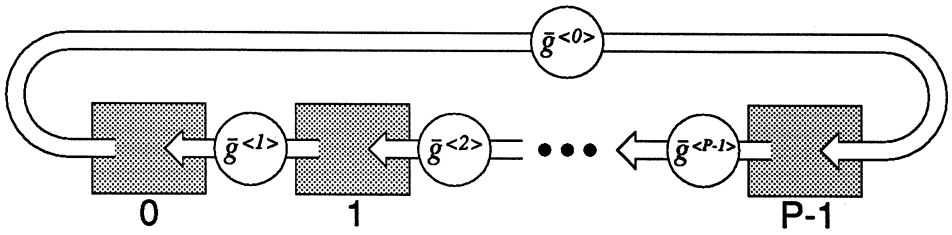

Figure 3.1: Summation of component gradients in a ring of processors

After $P-1$ steps each processor will have received component gradients from all other processors. Thus it will now hold the sum of all component gradients, and it will be able to update the weights as required. The OCCAM implementation of this communication scheme can be found in appendix B.5.

A similar scheme for the GF11 computer is described in [Witbrock], and apparently the same approach is used in [Bourrely] on the Hypercube.

An alternative and more efficient appi.tiach is also presented in [Witbrock]. The GF11 computer's communication facilities allow a much more complex inter-connectivity of processors than a ring. It is described how P processors may collect and sum component gradients in $O(\log_2 P)$ steps when configured in such a way that processor q is able to communicate with all processors $(q + 2^i) \bmod P$ for $i = 0, .., \log_2 P$.

Due to the heavy processor inter-connectivity, this approach cannot be applied to a system of transputers, each transputer having only four links by which it may be connected to other transputers. However, it is still possible to achieve the summing of component gradients in $O(\log_2 P)$ steps, since the transputers may be configured as a binary tree.[2]

[2]Ideed one might use a *trinary* tree, thereby utilizing all four available links on a transputer, yielding a time complexity of $O(\log_3 P)$. This would, however, merely reduce the height of the processor tree by a (small) constant factor, yet it would lead to a much more significant increase in memory size required for communicating the component gradients,

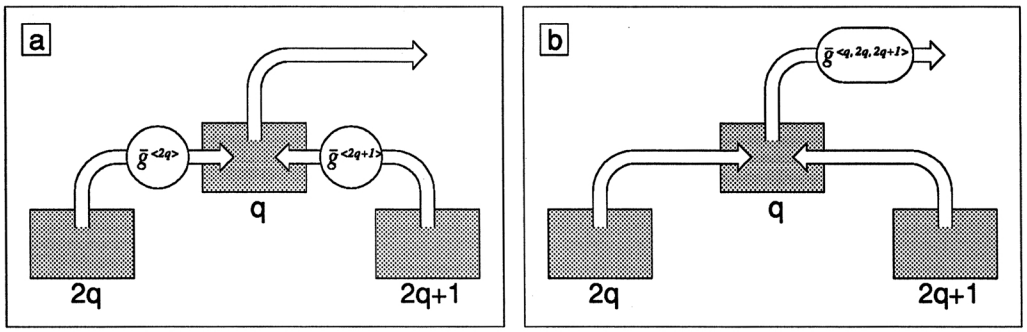

Figure 3.2: Summation of component gradients in a binary tree of processors

After all patterns in the current batch have been presented, all processors without successors in the tree structure send their accumulated component gradients to their parent processor. This is illustrated in figure 3.2a. All other processors collect the component gradients from their immediate successors, add those to their own, and send the result further upwards in the tree as shown in figure 3.2b. After $\lfloor \log_2 P \rfloor$ steps the root processor will hold the total gradient, which can then be used to calculate the weight changes. Using the tree structure again, the root processor can broadcast the updated weights to all other processors in $\lfloor \log_2 P \rfloor$ steps.

The OCCAM implementation of the data partitioning approach using a tree configuration of processors can be found in appendix B.4. The code segment for communicating component gradients and weights in the tree can be found in fold 29.

3.1.2 Comparison of Ring and Tree Configuration

P processors in a ring can update their weights in $P - 1$ steps, whereas it takes $2\lfloor \log_2 P \rfloor$ steps to perform the weight update in a tree configuration of P processors. This means, that when more than 5 processors are used the tree configuration requires less steps than does the ring configuration. So it seems obvious that a tree should be preferred when larger numbers of processors are used. However, contrary to the tree configuration the ring

see section 3.1.2.1. For this reason we have used only the binary tree configuration of processors.

allows the summations of component gradients to be performed concurrently with the sending and receiving of (other) component gradients. It should be noted, though, that performing communication and computation in parallel can at most save the time associated with the task requiring the smallest amount of time. Anyhow, the $P - 2$ steps of the ring where concurrent communication and summation is possible, must be smaller than each of the first $\lfloor \log_2 P \rfloor$ steps of the tree where communication and summation is performed sequentially.

We have made a number of test runs to determine whether or not this will mean that shorter updating times can be achieved with the ring configuration even when the number of processors exceeds 5.

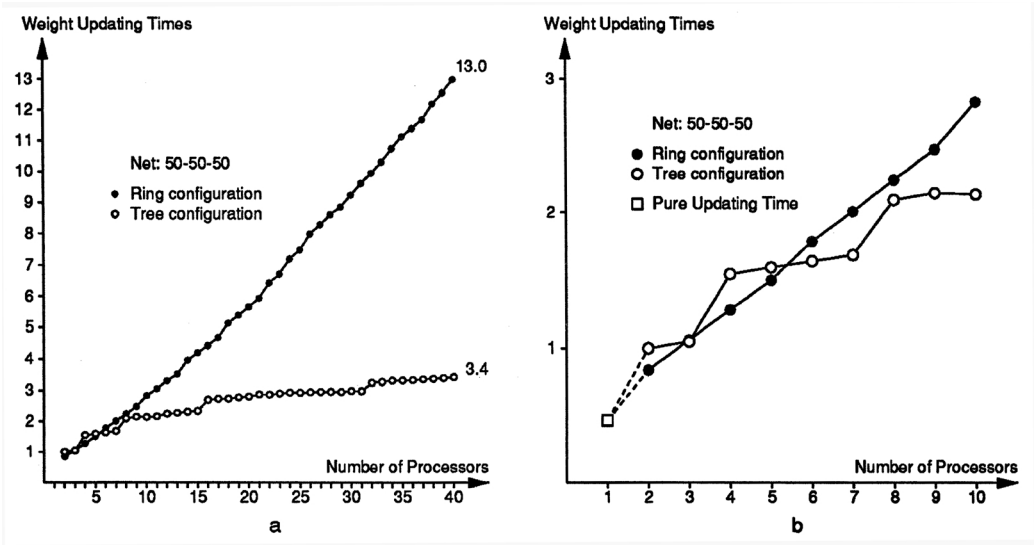

Figure 3.3: Time consumption in updating weights on different processor configurations

Figure 3.3a shows for different numbers of processors the time used for updating the weights in a 50-50-50 network using the ring and tree configurations. Updating times in the tree were measured in leaf-processors. All times have been normalized so that the time used for updating the weights in a tree of two processors corresponds to 1 on the y-axis.

As can be seen, both graphs have the expected shape. Weight updating times in the ring configuration increase in a roughly linear fashion with the

number of processors, whereas the time used in the tree configuration grows in a logarithmic manner. In the tree the updating time depends on the *height* of the tree: As long as adding one more processor to the tree structure does not increase the height of the tree, updating times are not increased (significantly) either. Only when the addition of a new processor increases the height of the tree, the updating time is increased somewhat.

The graphs show that, as expected, the ring configuration of processors is indeed capable of performing a weight update a bit faster than the tree configuration when the number of processors is 5 or below. For all larger numbers of processors the tree configuration is superior. This indicates that the time saved in summing the component gradients concurrently with communicating them must be relatively small. This is in accordance with the information found in appendix A where it can be seen that the time required to add two `REAL64` values is much smaller than the time required to communicate one `REAL64`.

Figure 3.3b shows the updating times for small numbers of processors. It can be seen that the constant contribution to the updating times observed in both the ring and the tree configuration arises from the actual updating of the weights after all component gradients have been collected. The time required for this marked with a square in the figure.

3.1.2.1 Memory Requirements

The memory requirements are nearly the same for both processor configurations. In both cases each processor needs to hold storage for two extra `REAL64` values for each weight in the net. These extra variables are necessary in the communication of component gradients.

The one difference with respect to memory requirements between the two algorithms is the following: In the ring configuration every processor needs to store the old weight changes in addition to the weight changes being calculated and the weights themselves. This is not necessary in the tree configuration, since the new set of weights are calculated in the root processor. Therefore only the root processor needs to store old weight changes. All other processors need only have storage capacity for the weights, the component gradients being calculated, and the two weight arrays for communication.

As each processor also keeps its own copy of all units in the network

we arrive at the following memory requirement[3] (measured in number of `REAL64`s) for each processor (except the root):

$$2 \cdot (NI + NH + NO) + 4 \cdot ((NI + 1) \cdot NH + (NH + 1) \cdot NO)$$

This very large memory requirement puts a much stricter limit than necessary on the size of neural networks that can be parallelized using the simple implementation of the data partitioning strategy. There are different ways of avoiding such large memory requirement. One is to avoid redundancy as much as possible, another is to refrain from communicating all of the component gradients at once. Both these strategies have been used in the algorithm discussed in section 3.2.

3.1.3 Performance of the Algorithm

In this section we are going to analyze to performance of the algorithm when run on nets of different size with varying batch sizes, using various numbers of processors. Since we have demonstrated that the tree configuration of processors is superior to the ring configuration whenever the number of processors exceeds 5 we will concentrate the following discussion of the simple implementation of the data partitioning strategy on the tree configuration.

It is worth noting at this point that *between* weight updates efficiency must be 100% since in those periods all processors work independently of each other, each processor running in exactly the same way as in the sequential neural net simulator. During weight updates, however, time may be wasted (in a computational sense) as the collection of component gradients and the distribution of the updated weights constitute extra work compared to the sequential algorithm. Furthermore, the computation of new weight values after all component gradients have been collected is not parallelized at all, since the root processor is performing this task alone. The inefficiency generated by this sequential computation of the weights increases with the number of processors: The more processors used in executing the algorithm, the more processors are inactive during the updating. It goes without saying that this is a very unfortunate property for a parallel algorithm, although the actual computation of the new weight values does not constitute a very large fraction of the total amount of computational work, especially when a large batch size is used.

[3]Only variables that scale with net size have been included.

To summarize: Since between weight updates the algorithm executed in each processor is exactly the same as the sequential algorithm we expect any reduced efficiency that may be observed in the parallel algorithm to be caused primarily by circumstances relating to the updating of the weights.

3.1.3.1 Effect of Varying the Batch Size

The above considerations suggest that efficiency will be low when a small batch is used, and that efficiency will increase if the size of the batch is increased.

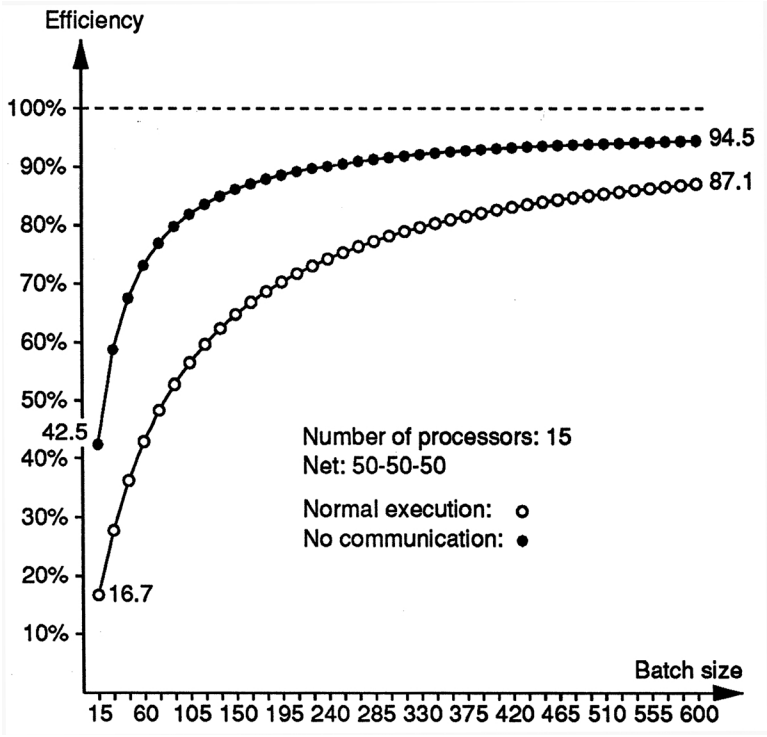

Figure 3.4: Efficiency shown as a function of batch size

Figure 3.4 shows the result of running 15 processors on a 50-50-50 network with varying batch size. The neural net simulator was run for 10 learning cycles, i.e. the weights were updated the same number of times in all runs. Note, that since the number of processors and the size of the net are fixed,

the time necessary for performing one weight update is expected to remain the same in all runs, independently of the batch size used.

The unfilled circles mark the results of ordinary executions of the algorithm, that is, when the processors actually communicate with each other as they should. The filled circles give the results of the same runs, only this time no communication is performed. Each time the processors would normally want to communicate they execute a `SKIP` command instead. By looking at the difference between those two graphs we should be able to estimate the importance of communication as a hindrance to optimal efficiency.
As expected, the efficiency of the ordinary algorithm is very low for a batch size of 15, since each processor in this case presents only one single pattern to the net between each weight update. In other words, the weights are updated relatively often. As the size of the batch is increased, execution time becomes more and more dominated by the time used for presenting patterns and calculating component gradients, since the number of weight updates is not increased. This way the time used for updating the weights becomes a smaller and smaller part of the total execution time, hence efficiency increases.

The second graph consisting of the filled circles confirms, that time spent in actual communication during each weight update is partly responsible for the less than optimal efficiency observed. Furthermore, absolute running times show that time spent in communication is independent of the batch size used, as expected.

However, the graphs also show that there must be other equally important sources for inefficiency. Some of this inefficiency may be generated by the extra `IF` sentences checking the number of successors, extra summations of component gradients, as well as the creation and termination of new processes handling communications.

Apparently, though, these are not the main reasons for the observed inefficiency of the non-communicating algorithm. By comparing the results of the parallel algorithm with those of the sequential algorithm (appendix B.3), we can see that the running time of the parallel algorithm without communication is less than 10% longer than that of the sequential algorithm, when the sequential algorithm is executed using the same number of patterns in the batch as the number of patterns handled by *each* of the processors in the parallel algorithm.[4] Therefore most of the observed inefficiency is

[4]In other words, the running time of the sequential algorithm using a batch size of b

probably due to the fact that because the number of weight updates is fixed, it does not take the sequential algorithm P times as long to handle P times as large a batch. Even though each of the processors in the parallel algorithm, between weight updates, runs like the sequential algorithm, it handles less patterns than does the sequential algorithm. Obviously, the effects of this phenomenon decreases as the number of patterns in the batch is increased. But even with as large a batch as 600 patterns, the effects are visible: If the parallel algorithm using 15 processors and a batch size of 600 were able to run as fast as the sequential algorithm on a batch of 40 patterns ($600/15 = 40$), it would only obtain an efficiency of 97.6%.

3.1.3.2 Effect of Varying the Number of Processors

We will now examine how well the algorithm performs when the problem is fixed and the number of processors is increased. Since increasing the batch size increases efficiency one would suspect that more processors can be used on problems with large batch sizes. Thus, we have carried out three independent experiment series using different fixed batch sizes.

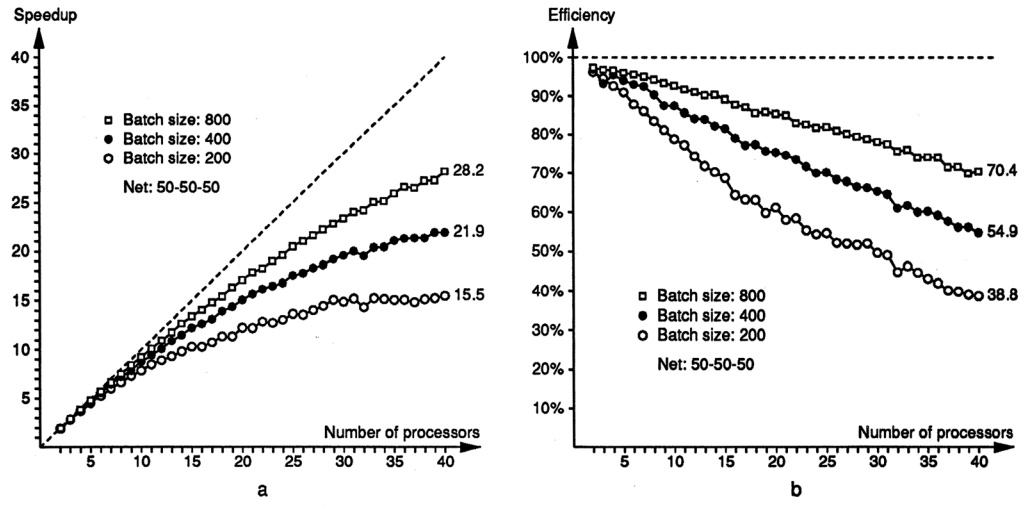

Figure 3.5: Speed-up for various numbers of processors and three different batch sizes

patterns is compared to the running time of the parallel algorithm using a batch size of $P \cdot b$ patterns.

Figure 3.5a shows how speed-up depends on the number of processors executing the parallel algorithm. It can be seen that for all three batch sizes (except, perhaps, the smallest batch size) speed-up has not yet reached its maximum value even when using 40 processors. This means that using still more processors would probably yield an even greater speed-up.

It can also be seen, though, that as more and more processors are used, efficiency decreases (figure 3.5b). Using the results just obtained in the above section 3.1.3.1 we can point out one reason for this: As the number of processors is increased each processor handles a smaller and smaller part of the batch. In other words, the batch size *per processor* is lowered, making the processors run less efficiently as demonstrated in section 3.1.3.1. This effect of batch size can also be seen by simply comparing the three graphs in figure 3.5.

However, there must be other reasons for the falling efficiency as well, since efficiency depends not only on the batch size per processor, but on the number of processors as well. This can be seen in figure 3.5b by comparing the points on the three graphs that represent the same *batch size per processor*. As an example, let us consider table 3.1 giving the efficiency of the three runs where each processor presents 20 patterns between weight updates.

Numbers of processors	Batch size		Efficiency
	Total	Per Processor	
10	200	20	78.8%
20	400	20	75.4%
40	800	20	70.4%

Table 3.1: Efficiency of three runs with 20 patterns per processor in the batch

The table shows that efficiency is reduced when more processors are used even though the batch size per processor remains constant (see also the following section 3.1.3.3). The source for this effect is probably the time needed for collecting component gradients and distributing new weights, since this time depends on the height of the processor tree. In fact, the reduction of efficiency following an increase in the height of the tree can be seen in figure 3.5. For batch size 200 and 400 a decrease in speed-up and efficiency is seen to be the result of increasing the number of processors from 31 to 32. Also, for batch size 200 a small reduction in efficiency is visible as the number of processors is increased from 15 to 16. The effects of such imbalances in the processor

tree should become less important as the batch size is increased (as is indeed observed to be the case), since it is only during weight updates that the imbalance influences the execution time.

Other small variations that can be observed in each of the graphs may be due to delays arising from differences in how large a part of the batch size each processor has got.

3.1.3.3 Scaling Batch Size with the Number of Processors

We will now examine the effects of scaling the size of the batch with the number of processors used in executing the algorithm. In the terms of chapter 2 we will investigate whether an increase in the number of sub-problems (patterns in the batch) will allow us to increase the number of processors correspondingly without reducing efficiency.

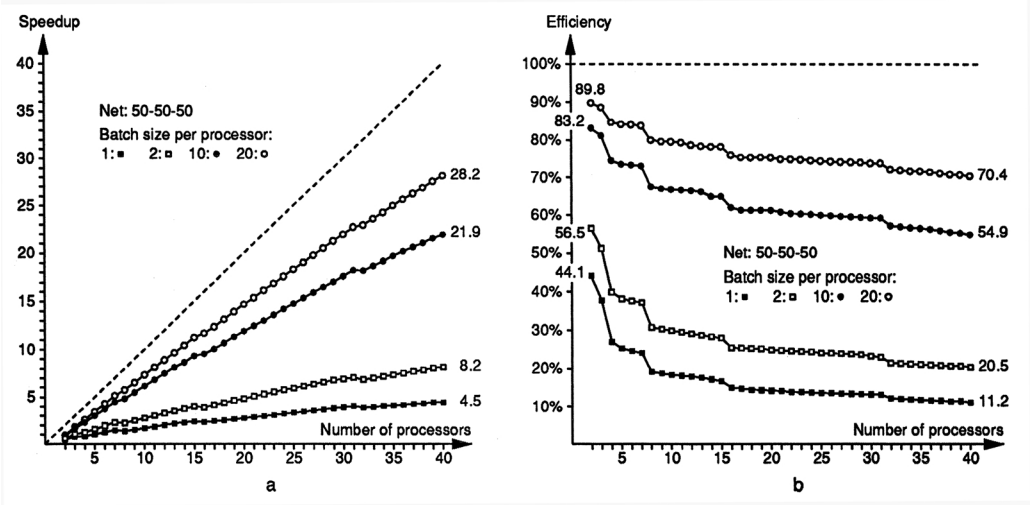

Figure 3.6: Efficiency shown as a function of batch size per processor

In figure 3.6 the batch size *per processor* is fixed (in each of the four series of experiments). That is, each processor presents a fixed number of patterns between each weight update, independently of the number of processors.

Figure 3.6a shows that we can obtain a nearly linear speed-up when the batch size per processor is fixed, However, it is clearly visible that speed-up is momentarily reduced whenever the height of the processor tree is increased.

Also, it seems that the rate with which speed-up grows becomes smaller as a result of the increased height.

This effect is even more distinct in the efficiency graphs shown in figure 3.6b. Every increase in the height of the processor tree reduces the efficiency of the algorithm. Notice, though, that as long as the height is unchanged, efficiency is (nearly) independent of the number of processors used. For large numbers of processors this means, that if increasing the batch size is possible (and acceptable with respect to learning capabilities, see chapter 5), the number of processors applied to the problem can be increased correspondingly without any significant loss of efficiency.

When compared to figure 3.3 it can be seen that the efficiency graphs in figure 3.6b are very similar to the graph showing the weight updating times in a tree configuration of processors. In fact, they are nearly identical in shape if the graph of figure 3.3 is turned upside down. This confirms what we have already stated several times, namely that it is mainly during weight updates that efficiency is lost in this parallel algorithm.

We can also use these figures to determine whether it will pay to apply extra processors to some fixed problem. In other words, for how long an increase in speed-up can be obtained by adding more processors. Raising the number of processors by some factor will only increase speed-up if efficiency is reduced by less than that same factor, as a result of the extra processors.

It would be nice if one could formulate a rule as to how many processors could be applied to some specific problem. Like, say, one should at most use a number of processors corresponding to one fifth of the batch size. That is, as long as each processor handles more than 5 patterns in the batch, adding more processors will increase speed-up. In general, this is possible if the efficiency of the parallel algorithm depends only on the batch size per processor, and not on the number of processors itself (i.e. if the efficiency graphs in figure 3.6b were straight, horizontal lines). As can be seen, this is not the case for this parallel algorithm, and indeed, the following considerations seem to indicate that no such general rule can be formulated.

As an example, let us assume that we are dealing with a fixed problem requiring the size of the batch to be 400. If we use 20 processors in simulating this neural net, each processor will handle 20 patterns in the batch, yielding a speed-up of about 15. We can also choose to use 40 processors, thereby letting each processor handle only 10 patterns. In this case, as shown in the figure, we get a speed-up of about 22. In other words, it pays to use the extra processors (as we have also seen in figure 3.5).

If 30 processors are used in dealing with a problem using batch size 600, each processor again has 20 patterns. But since we do not have 60 processors available, we cannot test whether letting each processor handle only 10 patterns in the batch will lead to an increase in speed-up. However, we have reasons to believe that the speed-up obtainable with 60 processors would indeed by higher than that which can be achieved with 30 processors. One reason is, that a speed-up similar to that achieved with 30 processors on a batch of 600 patterns is already obtained for 40 processors when each processor handles 10 patterns. And since the figure shows that speed-up increases in a nearly linear fashion when each processors takes care of 10 patterns, we do have reason to expect, that an even higher speed-up will be achieved with 60 processors and a total batch size of 600 patterns.

Now, the question is whether it will always pay to use as many processors as one tenth of the batch size. We saw, that with 40 processors, each handling 10 patterns, a speed-up was achieved similar to that obtained with 30 processors and 20 patterns per processor. In other words, even though the batch size per processor was halved, we only had to use 33% more processor in order to get the same speed-up. Yet the necessary increase in the number of processors is even smaller when we go from 20 to 10 patterns per processor, if we start out, with only 20 processors. In that case, we only have to add 5 processors, or 25% more processors, to achieve the same speed-up.

These figures seem to indicate, that the necessary increase itself becomes larger as more processors are used. In other words, the larger the number of processors, the larger a relative increase is necessary for the same speed-up to be achieved with half the batch size per processor.

If this tendency is extendable to larger numbers of processors, we will eventually reach some number of processors, where the increase must be more than 100%, if the same speed-up is to be achieved with 10 patterns per processor, as with 20 patterns per processor. When this happens, using only as many processors as one tenth of the batch size will be too many processors, yielding no gain in speed-up.

Similarly, we may observe, that it does not pay to use 40 processors (instead of 20) on a problem with batch size 40. Using 20 processors with each processor handling 2 patterns yields a speed-up of 5, whereas 40 processors with one pattern per processor yields only a speed-up of 4.5. In other words, the effect of adding extra processors to the tree is merely to slow down the whole system.

It is obviously a less desirable property of a parallel algorithm, that one

cannot double speed-up with twice as many processors on a problem containing twice as many subproblems. This is another reason behind the development of the advanced implementation of the data partitioning strategy described in section 3.2. By interleaving communication and computation we hope to be able to construct an algorithm whose efficiency depends only on the batch size per processor, and not on the number of processors itself.

3.1.3.4 Nets of Varying Size

It is very easy to determine whether efficiency will increase or decrease as a result of varying the size of the net. All that is necessary is to observe how the ratio between the number of weights and the amount of computation associated with the net varies with the size of the net. If the amount of computation does not grow as fast as the number of weights efficiency will decrease, because the time wasted in updating the weights is proportional to the number of weights.

The graphs on figure 3.7 are efficiency graphs showing the effects of varying the number of input, hidden, and output units (thereby varying the number of weights in the net). All runs were made using 15 transputers, and the number of learning cycles as well as the batch size remained constant throughout all simulations.

In figure 3.7a the effect of scaling up the entire net is shown. The networks used are n-n-n networks, i.e. there are equally many input, hidden and output units. Efficiency can be seen to decrease from 77.5% to 66.1% when n is increased from 1 to 10.

When trying to explain this decrease it is worth noting that (except for the bias weights) the number of weights grows quadratically with the number of units in an n-n-n net. This means that the amount of computation does not grow as fast as the number of weights does, since there is a substantial amount of work associated with calculating the activation function and its derivative in hidden and output units (see section 1.6.1). This computational work is only scaled up 10 times when scaling up the net from a 10-10-10 net to a 100-100-100 net, whereas the number of weights is increased with a factor of 100. Therefore the total amount of computational work associated with the net is not increased as much as the size of the component gradients communicated between processors during weight updates, which is why we observe a reduced efficiency.

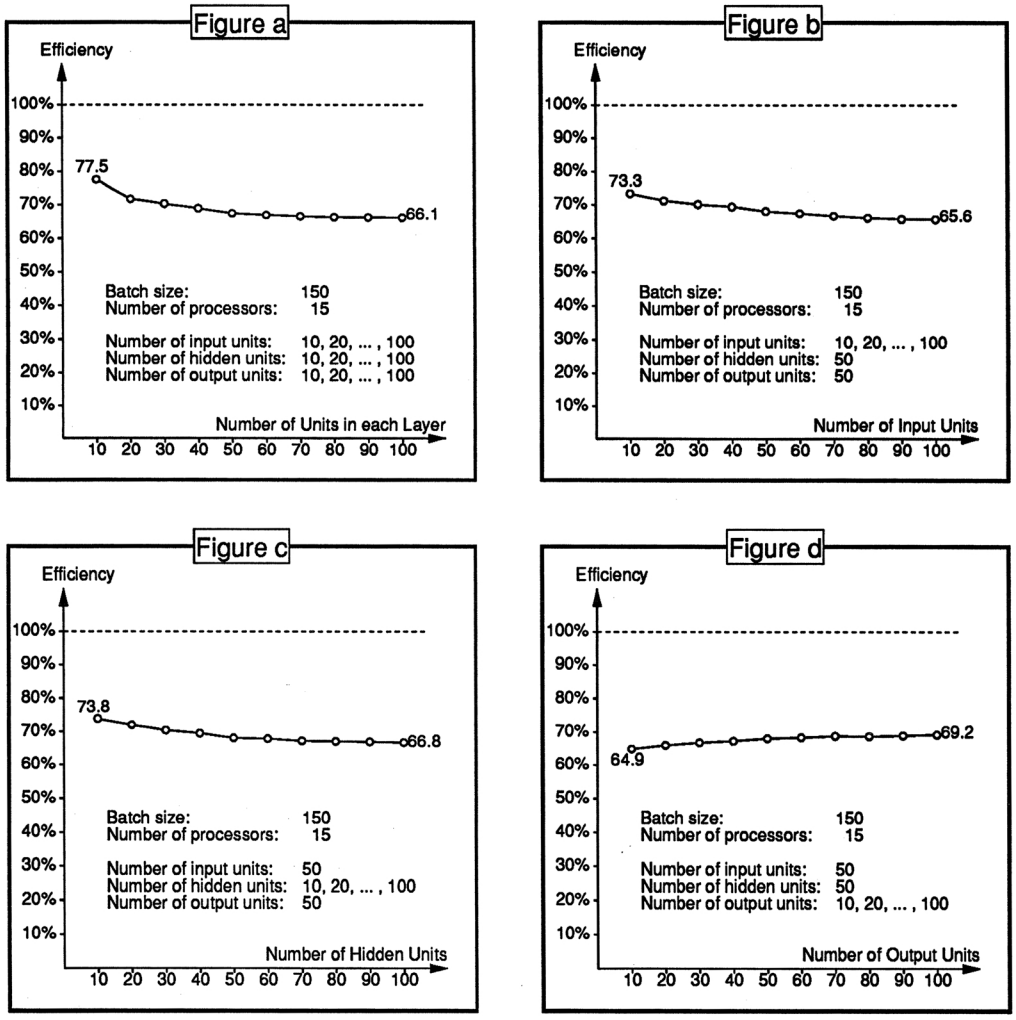

Figure 3.7: Efficiency shown as a function of network size

This is also the reason why the biggest reduction in efficiency takes place when rather small nets are scaled up a bit. When larger nets are scaled up the computational work associated with the units no longer form any significant part of the total amount of computation, which leads to a relatively smaller reduction in efficiency.

Figure 3.7b shows the effect of varying only the number of input units in the net. As can be seen the efficiency of the parallelization decreases when more input units are included in the net. The reason for this is that there

is no extra computation associated with the inclusion of extra input units except the calculations associated with the new weights between input and hidden units. And as can be seen in table 1.3 in section 1.6.1 the amount of computational work associated with each weight between input and hidden units is somewhat less than that associated with each weight between hidden and output units.[5] Therefore the size of the component gradients communicated is increased more than the computational work is, which leads to a smaller efficiency.

Adding extra hidden units to a net also reduces the efficiency a bit (figure 3.7c). Whenever the number of hidden units is increased, the number of weights in the net (and thereby the size of the component gradients communicated) is increased correspondingly. But the amount of computational work does not increase as much. All work associated with the output units (calculation of the activation function and its derivative) is unchanged, which must be why the efficiency is reduced when the number of hidden units is increased.

However, efficiency actually increases when extra output units are included as can be seen on figure 3.7d. This must be due to the fact that only the number of weights between hidden and output units is increased and, as mentioned before, a larger amount of computational work is associated with each of these weights than with each weight between input and hidden units. Since all added weights are associated with more computational work than the average weight the time necessary for computation is increased more than the time needed for updating the weights, leading to a better efficiency. Since there is no significant difference in how much computational work is required in the simulation of a hidden unit and an output unit, the amount of computational work associated with the units themselves is in reality scaled up like the number of output units.

3.1.4 Conclusion

We have seen that it is possible to parallelize the back-propagation algorithm in a very simple way using the data partitioning strategy. Furthermore, we have not surprisingly found that efficiency increases with the size of the batch of training patterns presented to the net between each update of the weights.

[5]This is true even if batch updating of the weights is used since updating the weights require equally many operations for the two layers of weights.

This means that if very large batch sizes can be used it is possible to gain a substantial speed-up using this kind of parallelizing. As for how learning is affected by large batch sizes, see chapter 5.

An important disadvantage of this algorithm is the somewhat excessive memory requirements resulting from the fact that each processor has to store a copy of the entire network. Such redundancy is not desirable in a parallel algorithm, since this means that the parallelization does not allow an increase in the size of the problems (with respect to memory requirements) that can be undertaken.

3.2 An Advanced Implementation of the Data Partitioning Strategy

In this section a less memory intensive implementation of the data partitioning strategy is described. Individual processors no longer keep their own copy of the entire network. The weights are stored in one processor only (called the *administrator*), and are circulated one by one between the processors whenever they are needed.

The administrator holds no training patterns, and does not itself present patterns to the neural network. This is done exclusively by the rest of the processors (referred to as the *slaves*), among whom training patterns are distributed evenly. The administrator controls the circulation of weights and performs the weight updates.[6]

Since each slave does not hold permanently any of the weights itself, it is essential that each weight when received is used in as many calculations as possible, so as to avoid unnecessary communications of weights. During the forward pass this means, that when weights connecting units in one layer to units in the next layer are communicated, they should be used for propagating the activity related to as many patterns as possible, i.e. processor q should use the weights to propagate the activity of *all* patterns in its share B_q of the current batch of patterns. In order to be able to do that, each slave processor q has to have as many copies of all units in the net as there are patterns in B_q.

[6]Actually, it turns out that the administrator *should* handle some patterns, although fewer than the slaves. For the moment, though, assume that the administrator does not present patterns to the net.

As a result of the above considerations all patterns in B_q are propagated forwards by processor q before any of the error values generated by those patterns are propagated backwards in the net. This is contrary to the algorithm described in section 3.1 in which one pattern was propagated forwards *and* backwards, and the resulting single pattern component gradient was calculated, before the presentation of the next pattern was handled.

The development of this algorithm was based on a short description of a similar algorithm constructed by Pomerleau et.al. [Pomerleau1] for the Warp computer.

3.2.1 Processor Topology

All slave processors need to receive every weight in the net once during the forward pass, starting with the weights between the input and the hidden units. Configuring the processors as a ring turns out to be an ideal solution. At first a ring structure may seem to be less than ideal, since the number of communication steps necessary for sending a weight from the administrator to some slave may be quite large. However, all slave processors need the same weights, but they do not necessarily have to make use of the same weight at the same time. Therefore the weights can simply be sent one by one through the ring in a pipelined fashion. This way no processor will have to wait for any weights except during the start-up and shut-down phases which constitute a very small part of the total running time since the number of weights is usually very much larger than the number of processors.

3.2.2 Handling Communication

Before describing in detail how the forward and backward passes are handled in this algorithm one point is worth observing. A transputer communicates via a set of four *links*. As stated in appendix A, there is associated with each of these links a small on-chip link processor handling all communication on the corresponding link, independently of the main processor. This means that not only can the transputer communicate simultaneously on several links, it can also to a very high degree perform computations using the main processor in parallel with data being communicated on the links (see appendix A.4).

As mentioned in the previous section, the weights of the neural net are circulated one by one through the ring of processors during the propagation of activity. This means, that during the forward pass each processor both has

to receive and send along each single weight in the entire neural network, i.e. each processor has to participate in twice as many communications as there are weights in the net. Associated with each of the weights communicated is some amount of calculation. When using such an enormous amount of communication, it is important to construct the algorithm so as to ensure that this calculation can be performed concurrently with the receipt and sending of weights. If, furthermore, the computation time is larger than the communication time, the communication of weights can take place without significantly slowing down the calculations performed by the slave.

Time otherwise wasted by the main processor while waiting for communication to finish has been put to good use.

3.2.3 The Forward Pass

During the propagation of activity the weights are circulated one by one through the ring of processors by the administrator, as illustrated in figure 3.8.

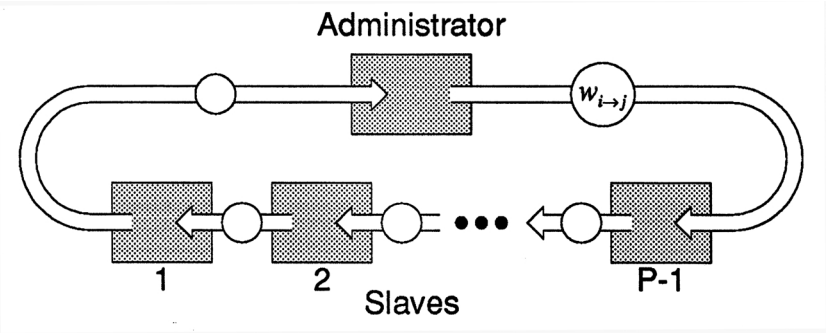

Figure 3.8: Circulation of weights during the forward pass

Initially, weights between the input layer and the hidden layer are circulated, thereby allowing the propagation of activity from the input units to the hidden units. Then weights between the hidden units and the output units are circulated, and the pattern of activity on the output units is calculated. These two steps of forward propagating activity are performed the same way, only difference being what weights are circulated and what units are used in the calculations. In the following we will therefore only describe in detail the propagation of activity from the input units to the hidden units.

Each time a slave processor q receives a weight $w^H_{i\to j}$ connecting unit i in the input layer to unit j in the hidden layer, it sends along the weight to the next processor in the ring (processor $q-1$). Concurrently with this, the weight is used to calculate the contribution of unit i to the net input of unit j for all patterns in this processor's share B_q of the total batch B. That is, for each pattern p in B_q the i'th part of the following sum is calculated upon receipt of $w^H_{i\to j}$:

$$\sum_{i=0}^{NI-1} a^I_{pi} w^H_{i\to j} = net^H_{pj}, \quad \forall p \in B_q \tag{3.3}$$

After all weights connecting the two layers have passed through the processors, each processor q contains for all units in the hidden layer the $|B_q|$ net inputs generated by the patterns in B_q. Each processor q then calculates for each pattern p in B_q the activation of all those units:

$$a^H_{pj} = f(net^H_{pj}) \ , \qquad \forall p \in B_q \ , \quad \forall j \in \mathcal{H} \tag{3.4}$$

Once the activity of the hidden units has been calculated it will be possible to determine the pattern of activity on the output units, This is done in exactly the same way as described above, except this time the administrator circulates the weights between the hidden units and the output units through the ring of processors.

The Occam implementation of the forward pass in the slaves can be found in appendix B.7 fold number 11. Notice, that the activity of the units in both the hidden and the output layer is calculated while the corresponding bias weights are circulated. This is done in order to increase the amount of calculation associated with the receipt of each bias weight, so that, hopefully, computation time will exceed communication time. Otherwise the only computations associated with each bias weight would be the calculation of the contribution of the bias unit to the net input of the recipient unit. Since the bias unit is an always-active, imaginary unit this requires only one addition, probably much too little computational work to take up all the time necessary for communicating the bias weights, at least when each processor handles only a few patterns.

3.2.4 The Backward Pass

The backward propagation of delta values and the computation of component gradients consist of 3 different steps.

3.2.4.1 Step One

Initially, delta values of all output units have to be computed for each of the patterns in the batch by the corresponding processors. For all patterns p in B_q processor q calculates the delta value for each output unit:

$$\delta_{pk}^{O} = a_{pk}^{O}(1 - a_{pk}^{O})(t_{pk} - a_{pk}^{O}) \ , \quad \forall p \in B_q \ , \quad \forall k \in \mathcal{O} \tag{3.5}$$

As can be seen this step requires no circulation of weights and can be performed independently by the various processors.

3.2.4.2 Step Two

The next step is the backward propagation of delta values to the hidden units. This step requires that the weights between hidden and output units are circulated again, in order to calculate the contribution of each output unit to the error of each hidden unit. This is done in a way similar to that described in 3.2.3 about the forward pass.

Each time slave processor q receives a weight $w_{j\to k}^{O}$ connecting unit j in the hidden layer and unit k in the output layer, it sends along the weight to the next processor in the ring (processor $q-1$). Concurrently with this, the weight is used to calculate for all patterns in B_q the contribution of output unit k to the error of hidden unit j. That is, for each pattern p in B_q the k'th part of the following sum is calculated upon receipt of $w_{j\to k}^{O}$:

$$\sum_{k=0}^{NO-1} \delta_{pk}^{O} w_{j\to k}^{O} = e_{pj}^{H}, \quad \forall p \in B_q \tag{3.6}$$

After all weights connecting hidden and output units have passed through the processors, processor q contains for all hidden units the $|B_q|$ error values generated by the patterns in B_q. Each processor then calculates for all patterns in B_q the corresponding delta values for all hidden units:

$$\delta_{pj}^{H} = a_{pj}^{H}(1 - a_{pj}^{H})e_{pj}^{H} \ , \quad \forall p \in B_q \ , \quad \forall j \in \mathcal{H} \tag{3.7}$$

3.2.4.3 Step Three

Step three is the calculation of the gradient resulting from the presentation to the net of all patterns in B.

The partial component gradients associated with the weights between input and hidden units are calculated the following way. Using equation 3.2 we find that the weight change $\Delta_B w^H_{i\to j}$ of weight $w^H_{i\to j}$ with respect to the patterns in batch B can be expressed as:

$$\begin{aligned}\Delta_B w^H_{i\to j}(n+1) &= -\eta \sum_{p\in B} \frac{\partial E_p}{\partial w^H_{i\to j}} + \alpha \Delta_B w^H_{i\to j}(n) \\ &= -\eta \sum_{q=0}^{P-1} g^{H<q>}_{i\to j} + \alpha \Delta_B w^H_{i\to j}(n) \end{aligned} \tag{3.8}$$

In other words, in order to update the weight $w^H_{i\to j}$ the administrator needs to know the sum of the *partial* component gradients $g^{H<q>}_{i\to j}$ of all slave processors. By using equation 1.10 it can be seen that such a partial component gradient can be calculated the following way:

$$g^{H<q>}_{i\to j} = \sum_{p\in B_q} \frac{\partial E_p}{\partial w^H_{i\to j}} = -\sum_{p\in B_q} \delta^H_{pj} a^H_{ip} \tag{3.9}$$

The calculation of each of those partial component gradients can be performed by the corresponding processor, since we already know both delta and activity values for all patterns $p \in B_q$ of all units in the net. The summation of partial component gradients is performed in a manner similar to the calculations of net inputs and hidden unit errors. In Stead of circulating the weights, however, the partial component gradients being calculated are circulated between the processors, as illustrated in figure 3.9.[7]

Initially, for each partial gradient $g^H_{i\to j}$ to be calculated the administrator sends out, the value 0 to processor $P-1$ (figure 3.9a). Concurrently with this communication processor $P-1$ calculates its contribution $g^{H<P-1>}_{i\to j}$. When the 0 arrives, the partial component gradient of processor $P-1$ is added to the 0, and then sent along to processor $P-2$ (figure 3.9b). While th is communication takes place, processor $P-2$ calculates $g^{H<P-2>}_{i\to j}$, and, upon receipt of $g^{H<P-1>}_{i\to j}$ from processor $P-1$, adds those two partial component gradients, and sends the result $g^{H<P-2,P-1>}_{i\to j}$ along to processor $P-3$ (figure 3.9c).

[7]Actually, in the prog ram it is the *negated* partial component gradients that are calculated and circulated through the ring. This way we avoid having to negate the total partial gradient when calculating the weight changes.

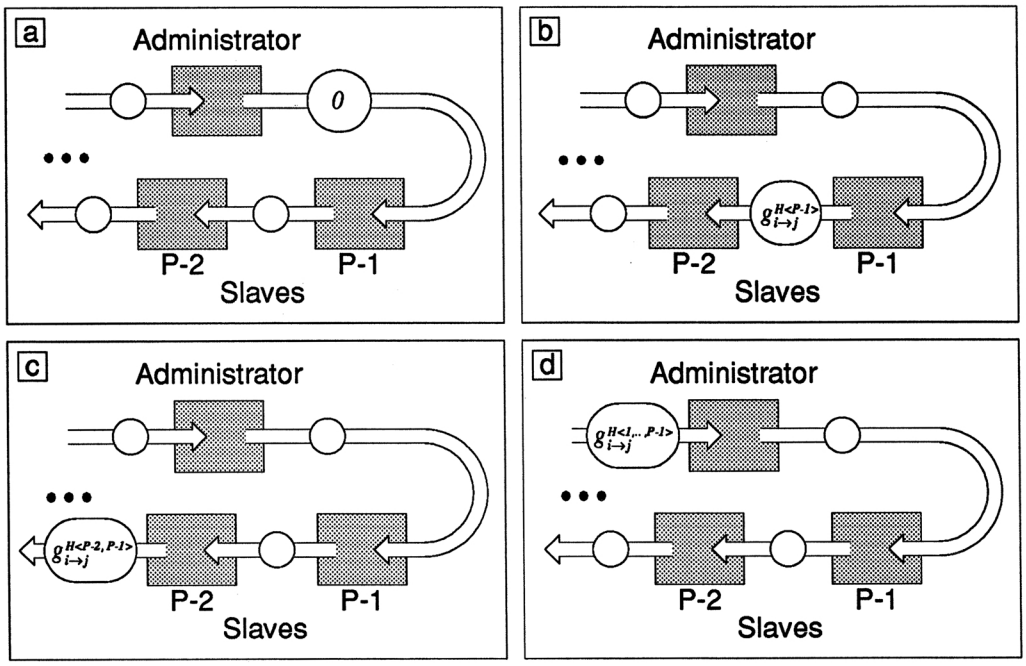

Figure 3.9: Computing partial gradients during the backward pass

In general, processor q calculates $g_{i\to j}^{H<q>}$ while, at the same time, receiving $g_{i\to j}^{H<q+1,\ldots,P-1>}$ from processor $q+1$. The partial component gradient calculated in processor q is added to this incoming partial component gradient and the result, $g_{i\to j}^{H<q,\ldots,P-1>}$, is sent to processor $q-1$ concurrently with the calculation of the next partial component, gradient (corresponding to the weight after $w_{i\to j}^{H}$).

This way partial component gradients are summed across all slave processors so that in the end (after P steps) the administrator will receive $g_{i\to j}^{H<1,\ldots,P-1>} = g_{i\to j}^{H}$, i.e. the partial gradient resulting from the presentation of all patterns in B (figure 3.9d). The administrator will then use this partial gradient to calculate the weight change $\Delta_B w_{i\to j}^{H}$ of weight $w_{i\to j}^{H}$.

The partial component gradients $g_{i\to j}^{O}$ associated with the weights between hidden and output units are calculated the same way.

When all partial gradients have been received and all weights updated, the administrator can send out the first weight between the input units and the hidden units, and the presentation of another batch of training patterns can commence.

The OCCAM implementation of the backward pass in the slaves can be found in appendix B.7 fold number 16. There are several ways in which the program is different from the algorithm described above. All of these changes were made in order to increase the amount of calculation associated with each communication.

- The calculation of error and delta values for the output units (step one) and the calculation of the partial component gradients related to the bias weights (part of step three) have been merged, so that when the delta value of output unit j is calculated so is the partial component gradient $g^{O<q>}_{bias \to j}$.
- Likewise, the calculation of hidden unit error values (first part of step two) has been overlayed with the calculation of the partial component gradients related to the weights between hidden and output units (part of step three). This means that upon receipt of $w^{O}_{j \to k}$ both the contribution of output unit k to the error of hidden unit j and the partial component gradient $g^{O<q>}_{j \to k}$ are calculated.
- The second part of step two (the calculation of hidden unit delta values) has been merged with the calculation of the partial component gradients related to the bias weights feeding into the hidden units.

It is not possible to merge the calculation of the partial component gradients corresponding to the weights between input and hidden units with extra computational work since the delta values of the hidden units are not propagated backwards to the input units.

3.2.5 Supplying the Administrator with a Share of the Batch

The administrator handles the circulation of weights, the collection of partial gradients, and the updating of the weights. Only the last of those tasks has any computational work associated with it. This means that during the forward pass the administrator will be idle most of the time, always waiting for the next weight to be sent or received. During the backward pass the administrator will have a little more to do, updating the weights as the partial gradients are received. However, there is one important difference between the work done by the administrator and the work done in the slaves: The

computational work in the slaves is proportional to the number of patterns in each slave's part of the current batch, whereas the computational work associated with updating the weights in the administrator is always the same, independently of the size of the batch.

The above considerations lead forth to the conclusion that time is probably wasted in the administrator when the batch per processor is large enough. There is an easy solution to this problem: Simply give the administrator its share of the batch – a number of patterns for it to present to the net concurrently with sending and receiving weights and partial gradients, and updating the weights.

What is not so easy is to determine how large a share of the batch the administrator should undertake. Obviously, the administrator should handle less patterns in the batch than the slaves, or it would have no time for the extra work it has to do. But giving patterns to the administrator will only increase efficiency if it relieves some of the pressure on the slaves, i.e. if it will make the slaves run faster. It will not help, though, simply to reduce the workload of some of the slaves, since in a ring the period between communications is determined by the processor communicating least often. This means that what the administrator should try to do is to reduce the workload of the slaves in such a way as to balance the workload evenly between the slaves. That is, the administrator should handle a number of patterns which will cause the slaves to have equally large shares of the batch, so that no slaves will have to wait for other slaves to become ready for communication.

On the other hand, if the administrator handles as many patterns as each of the slaves do, it will not have time for updating the weights. Or, rather, the administrator would delay all the slave processors, because they would have to wait for the administrator to finish the update of the weight. So therefore the administrator should always handle at least a few patterns less than each of the slaves. Since the time required for updating a weight is independent of the batch size, one would expect that the minimum difference between the number of patterns for the administrator and the number of patterns for each of the slaves should be some fixed number, corresponding to the time needed for updating one weight. However, experiments have shown that the administrator should at most handle about 96% or 97% of the number of patterns handled by each slave. In other words, although somewhat counter-intuitive the minimum difference between the size of the administrator's and the slaves' part of the batch should apparently depend

on how many patterns are actually handled by each slave. We have not investigated this peculiarity further, since it has only a very small influence on the efficiency of the algorithm, and mostly when the number of slaves is small. Usually, what decides how many patterns the administrator should handle is the considerations about all slaves handling equally large shares of the batch.

Experiments have also shown that if it is not possible for the administrator to take a number of patterns satisfying the above constraints then it should take no patterns at all.

3.2.6 Performance of the Algorithm

In this section we will describe the results of running the algorithm on different sets of parameters in order to determine the influence of various parameters on the efficiency of the algorithm.

3.2.6.1 Effect of Varying the Batch Size

Figure 3.10 shows the effect on efficiency of varying the size of the batch. As in figure 3.4 the unfilled circles mark the results of ordinary executions of the algorithm, that is, when the processors actually communicate with each other as they should, communication being performed concurrently with calculations. The filled circles give the results of the same runs, only this time no communication is performed. Each time the processors would normally want to communicate they execute a `SKIP` command instead. This means, that all extra assignments and processes necessary for communicating have not been removed. The filled squares show the efficiency obtained, if communication is not performed concurrently with computation, i.e. if calculations begin only after all communication is finished.

The graphs look very similar to those in figure 3.4. Efficiency is low for small batch sizes where each processor handles only a few patterns. The more patterns per processor the higher efficiency obtained. The graph consisting of the filled circles confirms, that communication time is partly responsible for the less than optimal efficiency observed, although for large batch sizes there is no large difference between the efficiency obtained when no actual communication takes place and the efficiency obtainable when communication is performed concurrently with calculations.

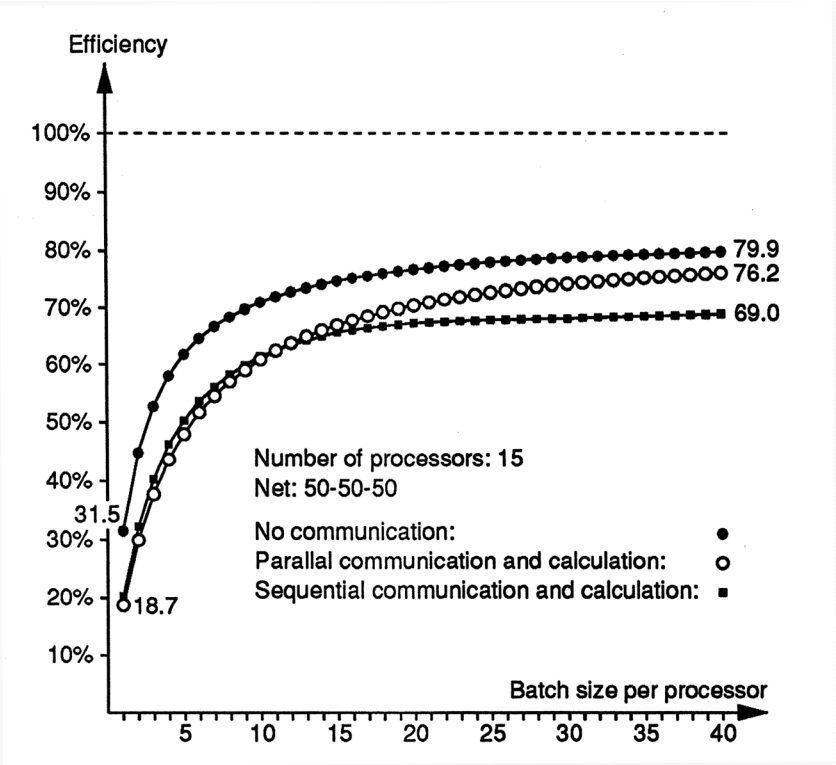

Figure 3.10: Efficiency shown as a function of batch size

By comparing the graphs we can see, that for small batch sizes it is actually a bit more efficient to perform communication and computation sequentially. The reason for this is probably, that for small batch sizes there is not enough computation to take up the time necessary for communicating the weights. So performing communication and computation concurrently does not save much time. Instead, time is wasted because of the extra work necessary for handling parallel communication and computation. However, the effect of performing communication and computation in parallel shows for batch sizes above 11 or 12 patterns per processor.

The graphs also show that there must be other important sources of inefficiency. Some of this inefficiency may be generated by extra assignments necessary for communicating in parallel with calculations, extra summations of component gradients, as well as the creation and termination of new processes handling communications.

3.2.6.2 Effect of Varying the Number of Processors

Figure 3.11 shows for three different batch sizes how speed-up varies with the number of processors used. For all three batch sizes it can be seen that speed-up apparently has not yet reached its maximum value even with 40 processors executing the algorithm. Adding more processors to the ring should therefore further increase the speed-up.

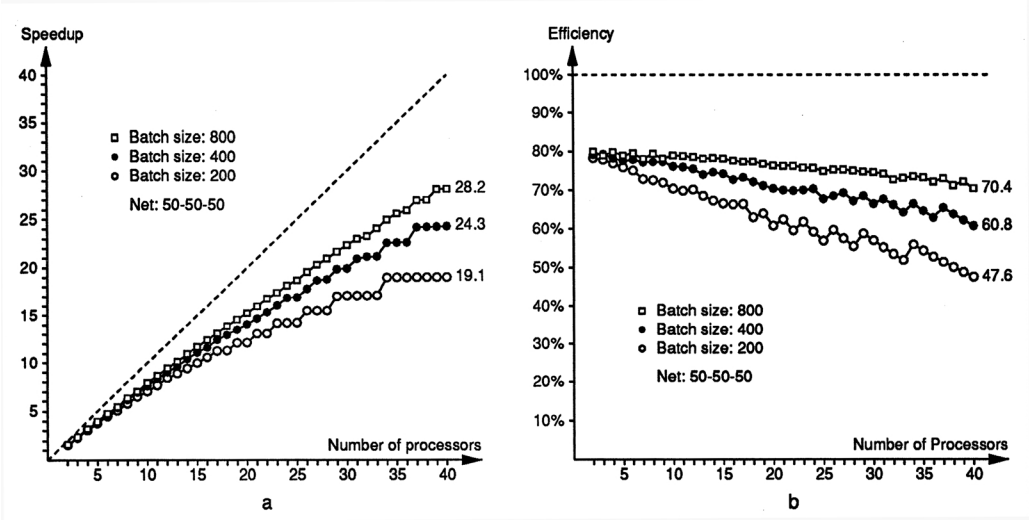

Figure 3.11: Speed-up for various numbers of processors

It can be seen, though, that at least for the two smallest of the batch sizes speed-up is not always increased by adding just one more processor. This is especially so for the graph showing the speed-up obtained with a batch of 200 patterns: No extra speed-up is gained in increasing the number of processors from 34 to 40. Likewise, no extra speed-up is obtained from increasing the number of processors from 29 to 33. But adding just one more processor to the 33 already in the ring produces a significant rise in speed-up. This confirms our claim in section 3.2.5 that in a ring the pace is set by the slowest processor: When between 29 and 33 processors are used some of the slaves in the ring have 7 patterns in their part of the batch, but with 34 processors the administrator can take 2 patterns for itself, thereby leaving exactly 6 patterns for each of the slaves.

We therefore have reason to suspect that an increase would occur again if 41 processors were available, since the 200 patterns could now become

distributed in such a way that all slaves would have precisely 5 patterns in their part of the batch. With 40 processors, 34 of the slaves have 5 patterns and 5 have 6 patterns, and the administrator handles no patterns.

On the other hand, we also have to realize that when using a batch of 200 patterns we cannot expect speed-up to become much higher after that, since speed-up can only jump four more times. Just after the last jump each slave will have exactly one pattern in its part of the batch, making the addition of more processors senseless. Naturally, as can be seen in the figure, for larger batch sizes the phenomenon described above occurs later, that is, for larger numbers of processors, since the relevant factor is the batch size *per processor*.

If figure 3.11a is compared to figure 3.5a on page 47 it can be seen that the advanced data partitioning parallelization is superior for the two smallest of the three different batch sizes, whereas, by some curious coincidence, the speed-up attained with 40 processors is identical in the two algorithms. Also, it seems that the smaller the size of the batch the larger is the difference between the two algorithms. This indicates that the advanced implementation should be used when relatively small batch sizes are preferred.

By comparing figure 3.11b with figure 3.5b we can see, that the efficiency graphs of the advanced implementation are very different in shape from those of the simple implementation of the data partitioning strategy. The efficiency of this algorithm is lower for small numbers of processors, but higher for large numbers of processors. Indeed, the shape of the graphs seem to suggest that efficiency is not reduced because the inclusion of extra processors slows down some part of the algorithm. The graph representing the results obtained with batch size 800 is nearly a straight, horizontal line, indicating that as long as the size of the batch is large enough, the addition of extra processors will not reduce efficiency significantly. This is confirmed by looking at the one point in each graph representing a batch size per processor of 20 patterns (see table 3.2). Apparently, efficiency depends only on the number of patterns per processor in the batch, not on the number of processors itself.

Efficiency is reduced because using more processors does not always reduce the number of patterns in all slaves, and because even if it does, figure 3.10 shows that less patterns for each slave leads to a lower efficiency.

Numbers of processors	Batch size		Efficiency
	Total	Per Processor	
10	200	20	70.37%
20	400	20	70.44%
40	800	20	70.41%

Table 3.2: Efficiency of three runs with 20 patterns per processor in the batch

3.2.6.3 Scaling Batch Size with the Number of Processors

In this section we will further examine the effects of scaling the batch size with the number of processors. Ideally, twice as large a speed-up should be obtainable with twice as many processors on a problem containing twice as many sub-problems.

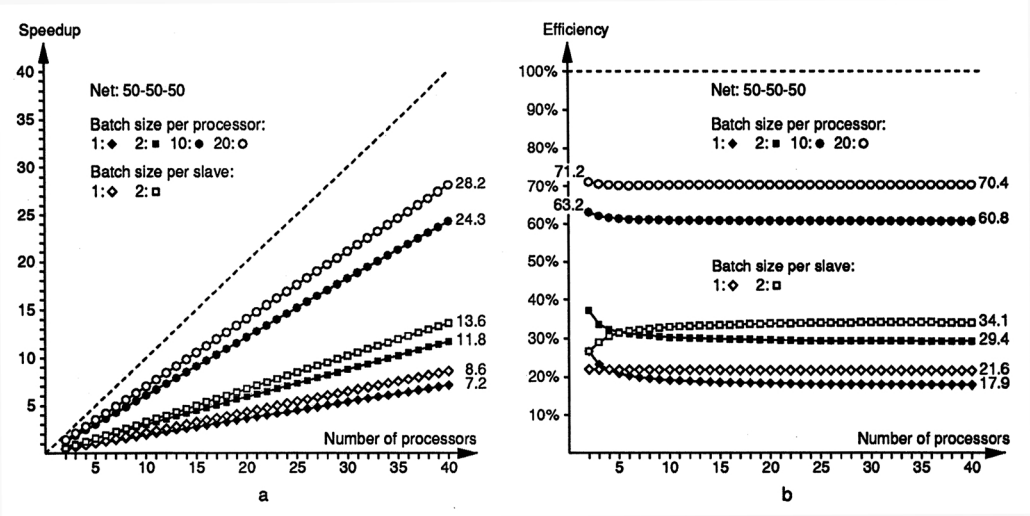

Figure 3.12: Efficiency shown as a function of batch size per processor

In four of the experiment series shown in figure 3.12 the batch size *per processor* is fixed. When batch size per processor is small, the administrator cannot handle any patterns at all. For a batch size per processor equal to one, this means that no matter how many processors are used, there will always be one slave handling 2 patterns instead of one. This slave will com-

municate with a frequency determined by how long time it takes to handle two patterns, thereby slowing down all other slaves. To be able to examine the effect of this, we have also made a couple of experiment series in which batch size is scaled with the number of *slaves*.

Figure 3.12b shows that efficiency is independent of the number of processors used in executing the algorithm, as long as the batch size per processor (or slave) is fixed. Therefore, in this case we are able to obtain a linear speed-up in the number of processors, as shown in figure 3.12a.

By comparing the different graphs in figure 3.12b we can see, that efficiency depends only on how many patterns each processor presents to the net during a learning cycle. The efficiency obtained with different numbers of patterns in the batch per processor agrees with the results shown in figure 3.10. In other words, because the efficiency graphs in figure 3.12b are straight, horizontal lines, figure 3.10 can be used as an indicator of the efficiency that can be obtained with a certain batch size per processor, independently of how many processors are used.

Also, since the efficiency graphs in figure 3.12 are straight, horizontal lines we may conclude, that whenever the size of the batch can be increased the number of processors may be increased correspondingly, without any loss of efficiency.

By comparing the speed-up graphs representing runs with 1 (or 2) patterns per *processor* with those with 1 (or 2) patterns per *slave*, it is obvious that for some fixed batch size, there will be a large gain in adding one more processor, so that instead of there being 1 (or 2) patterns per processor, there will be 1 (or 2) patterns per slave in the batch. This way all slaves will handle the same number of patterns, and no slave will slow down other slaves. Going from 1 pattern per processor to 1 pattern per slave by adding a processor will cause the last possible of the jumps in speed-up mentioned in section 3.2.6.2.

Note, that the speed-up graphs confirm what we stated in section 3.2.6.2 about the slowest slave setting the pace in a ring of processors. With 21 processors, 2 patterns per slave (yielding a batch of 40 patterns) results in a speed-up of 7.13, Using the same batch size and increasing the number of processors to 40 will result in a speed-up of no more than 7.16, because there is still one slave handling 2 patterns. In other words, the pace is set by the slave having the largest workload.

Notice also, that the efficiency that can be obtained using a batch of one pattern per slave is more than half the efficiency resulting from having

two patterns per slave. Therefore, if we assume that efficiency is preserved for arbitrary numbers of processors as long as the batch size per processor remains fixed, then we are able to conclude, that for all batch sizes the maximum speed-up is obtained by running the algorithm on as many processors (plus one, the administrator) as there are patterns in the batch. In other words, with regard to speed it pays to use as many processors as possible, until each slave processor handles only one pattern in the batch.

For some fixed size of the batch B, using P processors where $2(P-1) = |B|$ will yield a speed-up of about $0.34 \cdot P$, since efficiency with 2 patterns per slave is 34.1%, as it can be seen in figure 3.12b. The figure also shows that with one pattern per slave in the batch an efficiency of 21.6% can be obtained. This means, that if $2 \cdot P - 1$ processors are used (so that $P - 1 = |B|$) this will yield a speed-up of $0.216 \cdot 2 \cdot P = 0.432 + P$, i.e. a larger speed-up than with P processors.

Actually, by assuming that the efficiency graphs will continue to be straight, horizontal lines for even larger numbers of processors than are available to us, we are able to estimate the maximum obtainable speed-up for some arbitrary batch size. If the above considerations hold for as many as 801 processor then we may expect to achieve a maximum speed-up of $0.216 \cdot 801 = 173$ for a batch size of 800 patterns.

3.2.6.4 Nets of Varying Size

We will now examine whether the efficiency of this parallelization is sensitive to the size of the neural network simulated.

In section 3.1.3.4, we mentioned that the amount of computational work does not grow as fast as the number of weights when an n-n-n net is scaled up, since the work associated with calculating the activation function and its derivative is proportional to the number of hidden and output units, whereas the number of weights (excluding bias weights) grows quadratically with the number of units in the net.

This suggests that efficiency might be reduced as a result of increasing the size of the network, since the amount of computation is not increased as much as the number of communications. This is not the case, though, as illustrated in figure 3.13. Efficiency is seen to be nearly completely independent of the size of the net.

It is worth observing, that in this particular algorithm the calculation of the activation function and its derivative is associated with the bias weights

of the net (see section 3.2.3). In this context, the fact that the amount of computation is not increased as much as the number of communications does not mean, that the amount of work associated with one communication is lowered *in general*. It simply means, that in the 100-100-100 net there are relatively fewer bias weights, and since each bias weight has associated more computational work with it, than a weight connecting units in different layers has, the total amount of computational work has not grown as much as the number of weights.

In other words, there are simply more ordinary weights in the net, weights with less computational work associated. This does not necessarily lower the efficiency. If the time required for communicating one such weight is shorter than the time required for performing the computations, then performing these two tasks concurrently will make the cost of communicating insignificant. This is most likely the reason why efficiency is not reduced notably when the size of the net is increased.

3.2.7 Memory Requirements

One of the reasons for designing this algorithm was the excessive memory requirements of the algorithm described earlier in section 3.1. In that algorithm each processor kept its own copy of the entire set of weights. The design of the algorithm described in this section was based on the attempt of reducing memory requirements by avoiding the redundancy introduced by having to store P identical copies of all weights. In this algorithm only the administrator keeps the entire set of weights, whereas all other processors only need the capacity to store two or three weights. However, each (slave) processor needs to store several copies of all *units* in the network, since the complete presentation of one pattern (including forward propagation of activity and backward propagation of error) cannot be accomplished in one step (see the introduction to section 3.2). Hopefully, this extra memory demand is not nearly as large as the amount of memory saved by removing the redundant set of weights in each processor.

In the following we will concentrate on the variables that scale with either net size or batch size, i.e. we will exclude from consideration all variables that are constant in size in all runs. The maximum number of patterns in one slave processor's part of a batch is $\lceil B/(P-1) \rceil$, since it may not be possible to let the administrator participate in the presentation of patterns (see section 3.2.5). Therefore we get the following memory requirement for each slave

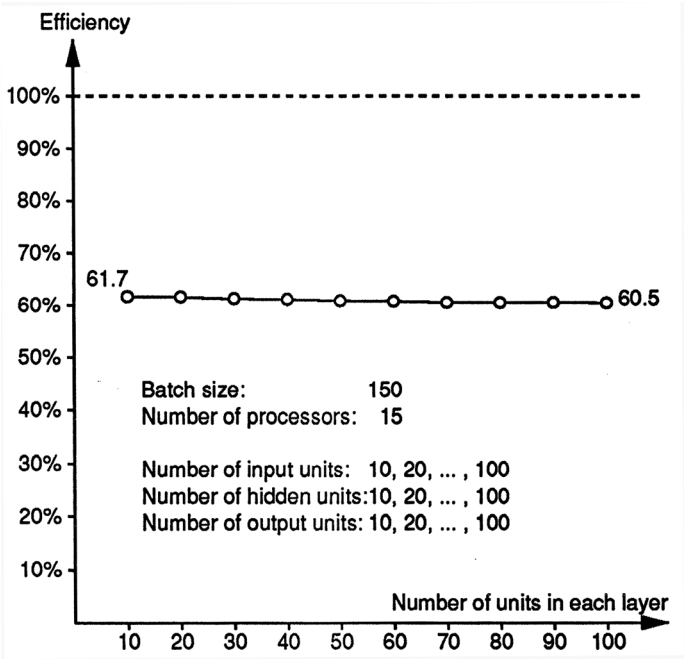

Figure 3.13: Efficiency shown as a function of network size

processor (compare with section 3.1.2.1).

$$\lceil B/(P-1) \rceil \cdot 2(NI + NH + NO)$$

The memory requirements of the slaves can be seen to be independent of the number of weights in the network, contrary to what was the case with the first data partitioning algorithm described. This is fortunate since the number of weights in a network grow quadratically with the number of units when the entire net is scaled up. Therefore the algorithm described in this section will be less memory intensive when run on large nets, as expected.

Using the formulas it can be seen that even on a rather small net like a 20-20-20 net this algorithm will be more economic with respect to memory as long as the batch size per processor is less than 28.

As long as the batch size is relatively small and the network is relatively large this algorithm will not require as much storage capacity (in the slaves) as the simple implementation of the data partitioning strategy. If there is too little memory in the slaves one solution is simply to use more slaves. This will most likely reduce efficiency a bit, though judging from the results of section

3.2.6.2 not enough to actually increase the running time of the algorithm. This is contrary to the simple implementation in which not much can be done if the memory requirement of some application is too heavy, except, of course, to change the algorithm so that only a part of the weights and the partial component gradients are communicated at one time, i.e. to split the weights and the partial component gradients into separate packets which can be communicated one after another. This does not, however, change the fact that in that algorithm there has to be a copy of the entire set of weights in each processor.

On the whole, the advanced implementation is preferable with respect to memory requirements, since the memory requirement of individual slaves is reduced as more processors are used.

3.2.8 Conclusion

We have seen that, with respect to both memory requirements and running time, this implementation allows larger problems (with respect to the number of sub-problems) to be handled when more processors are used.

An important property of this algorithm is that efficiency depends only on the number of sub-problems per processor, not on the number of processors itself. This is contrary to the simple implementation of the data partitioning strategy in which efficiency is reduced when more processors are used, even though the number of sub-problems per processor is not reduced.

The fact that efficiency does not depend on the number of processors in the advanced implementation means, that a problem containing twice as many sub-problems can be solved using twice as many processors without any reduction in efficiency. In other words, very large numbers of processors can be utilized when large problems are handled.

We have seen that this algorithm is faster than the simple implementation on relatively small batch sizes like 200 and 400 patterns, and that the two algorithms reach the same maximum speed-up on an 800 pattern batch. We have also seen that within the limits of the number of processors available to us, the highest speed-up is achieved when the maximum number of processors is used, i.e. when each slave processor handles only one sub-problem. When this is taken into consideration together with the apparent non-dependency of efficiency on the number of processors, we have reason to suspect that in terms of speed this algorithm would be able to out-perform the simple implementation on any batch size, if enough processors were available.

The above considerations lead forth to the conclusion, that this algorithm should be preferred, if, compared to the batch size, a relatively large number of processors is available. However, on problems where the batch size per processor is very large (more than 50 patterns) the simple implementation is probably going to achieve a higher speed-up.

3.3 An Implementation Using Matrix Operations

In this section a third parallelization of back-propagation will be described. The algorithm is more or less a combination of the data partitioning strategy and the net partitioning strategy. Even so, we have included the algorithm in this chapter. We have done this because, in one very important aspect, the algorithm is similar to the other two algorithms described in this chapter: It has to use batch updating of the weights.

The main idea of this algorithm is the observation that the basic calculations involved with back-propagation can easily be expressed as matrix operations (matrix multiplications). As algorithms for matrix multiplications on distributed-memory processors (as the transputers) are in existence already, one can simply use one of these algorithms and extend it with the neural net specific calculations.

This approach has been investigated by Petrowski et.al. [Petrowski]. They have implemented an optimal matrix multiplication algorithm by Fox et.al. [Fox2] on a system of 16 transputers. They use both torus and mesh configurations. These configurations (each with an additional root processor) are illustrated in figure 3.14.

The matrix multiplication parallelization does not observe one of the basic demands for parallelizations which we put forward in section 2.3: The torus and mesh configurations have to be *squares*, i.e. equally many processors in the rows and columns. Hence, there are not very many numbers of processors that can be used.

We have included this algorithm because the results reported in the article suggest that the implementation by Petrowski et.al. is superior in some aspects to the two implementations in sections 3.1 and 3.2. However, as our own implementation of the matrix multiplication strategy will show,

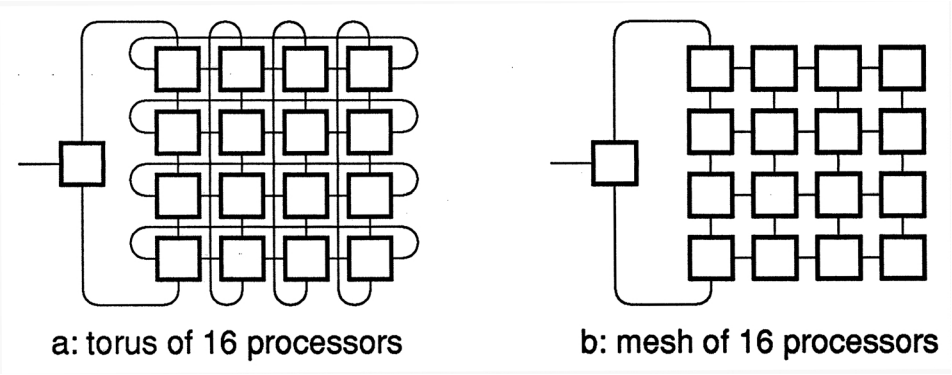

Figure 3.14: The torus and mesh configurations with additional root processor

this superiority probably relies on certain "tricks" which we have not used in the other implementations.

Therefore, our analysis of this algorithm will not be as thorough as the analysis of the other algorithms. We will mainly explain the idea of the algorithm and compare our results with those obtained by Petrowski et.al.

3.3.1 The Use of Matrix Multiplications

In order to be able to describe this algorithm, we will have to extend our notation a bit. By $\overline{\overline{w}}^H$ we denote the NH by NI matrix of input to hidden unit weights:

$$\overline{\overline{w}}^H = \begin{pmatrix} w^H_{0\to 0} & \cdots & w^H_{NI-1\to 0} \\ w^H_{0\to 1} & \cdots & w^H_{NI-1\to 1} \\ \vdots & \ddots & \vdots \\ w^H_{0\to NH-1} & \cdots & w^H_{NI-1\to NH-1} \end{pmatrix} \tag{3.10}$$

Likewise, the NO by NH matrix of weights between hidden and output units is denoted $\overline{\overline{w}}^O$.

The algorithm described in the following is a parallelization of the batch updating algorithm. Hence, several patterns (the patterns of batch B) will be presented to the net, simultaneously. The matrix of input unit, acCvit,ies (input patterns) for a given batch, B, will be denoted by $\overline{\overline{a}}^I_B$. This is an NI

by $|B|$ matrix:

$$\overline{\overline{a}}^{I}_{B} = \begin{pmatrix} a^{I}_{0,0} & \cdots & a^{I}_{0,|B|-1} \\ a^{I}_{1,0} & \cdots & a^{I}_{1,|B|-1} \\ \vdots & \ddots & \vdots \\ a^{I}_{NI-1,0} & \cdots & a^{I}_{NI-1,|B|-1} \end{pmatrix} \tag{3.11}$$

The matrices of hidden and output unit activities are defined similarly. By $\overline{\overline{\delta}}^{H}_{B}$ and $\overline{\overline{\delta}}^{O}_{B}$ we denote the matrices of hidden and output unit delta values, respectively. These matrices have the same dimensions as the corresponding activity matrices.

The propagation of activity can be expressed as a matrix multiplication. The product, of the input to hidden unit weight matrix with the matrix of input unit activities results in an NH by $|B|$ matrix of hidden unit net input values[8] from which the matrix of hidden unit activities is easily calculated. In short we can write:

$$\overline{\overline{a}}^{H}_{B} = f(\overline{\overline{w}}^{H} * \overline{\overline{a}}^{I}_{B}) \tag{3.12}$$

Notice, the activation function f is invoked on every individual entry in the matrix.

The matrix of output, unit activities is calculated similarly. This matrix is then used in the calculation of output unit delta values (the matrix $\overline{\overline{\delta}}^{O}_{B}$). No matrix operations are involved here. Please refer to equation 1.8 (page 13) for the exact computations needed. The matrix of hidden unit, error values (used in the calculation of hidden unit delta values, $\overline{\overline{\delta}}^{H}_{B}$ is calculated as follows:

$$\overline{\overline{e}}^{H}_{B} = \quad {}^{t}\overline{\overline{w}}^{O} * \overline{\overline{\delta}}^{O}_{B}) \tag{3.13}$$

where the ${}^{t}\overline{\overline{w}}^{O}$ is the transposed matrix of hidden to output unit weights. Again, the delta values are easily computed when the error values have been calculated (see equation 1.9 on page 14).

At this stage the activity and delta values of all units for all patterns in the batch are known. It is then possible to calculate both the individual component gradients arising from the patterns in the batch B and their sum

[8]Notice, that the contribution from the bias weights is temporarily ignored.

in a single matrix multiplication. The part of the total gradient belonging to the weights between hidden and output units is calculated as:

$$\sum_{p \in B} \frac{\partial E_p}{\partial \overline{\overline{w}}^O} = \overline{\overline{\delta}}_B^O * {}^t\overline{\overline{a}}_B^H \tag{3.14}$$

where ${}^t\overline{\overline{a}}_B^H$ is the transposed matrix of hidden unit activities. The calculation of the part of the total gradient for the input to hidden unit weights is performed similarly.

When the total gradient has been calculated, the actual weight changes are easily computed (see equation 1.12 on page 15).

3.3.2 The Parallelization

Three different types of matrix multiplications have been identified in the back-propagation algorithm: The multiplication of equation 3.12 is used twice, once when the calculation of hidden unit activities is performed and once again in the calculation of output unit activities. Equation 3.13 is used once in the calculation of hidden unit delta values. Finally equation 3.14 is used twice when calculating the total gradient.

In some of these matrix multiplications, one of the matrices has to be transposed. As all matrices are partitioned among the processors, this is a very inefficient operation if the processors are not configured as a *square* mesh or torus, i.e. the number of processors in a row and in a column are the same. This effectively puts a limit to the number of processors that can be used.

As mentioned, the weight, activity, and delta matrices are partitioned evenly among the processors, i.e. each processor does not hold a copy of everything. The partitioning is done as follows: If Q by Q processors are used (Q is the square root of the number of processors) each matrix is divided into Q^2 sub-matrices and distributed to the right processors. E.g. the NH by NI matrix of weights between input and hidden units is divided into sub-matrices of size $\frac{NH}{Q}$ by $\frac{NI}{Q}$. When 16 processors are used we have the

following division of the matrix:

$$\overline{\overline{w}}^H = \begin{pmatrix} \overline{\overline{w}}^H_{[0,0]} & \overline{\overline{w}}^H_{[0,1]} & \overline{\overline{w}}^H_{[0,2]} & \overline{\overline{w}}^H_{[0,3]} \\ \overline{\overline{w}}^H_{[1,0]} & \overline{\overline{w}}^H_{[1,1]} & \overline{\overline{w}}^H_{[1,2]} & \overline{\overline{w}}^H_{[1,3]} \\ \overline{\overline{w}}^H_{[2,0]} & \overline{\overline{w}}^H_{[2,1]} & \overline{\overline{w}}^H_{[2,2]} & \overline{\overline{w}}^H_{[2,3]} \\ \overline{\overline{w}}^H_{[3,0]} & \overline{\overline{w}}^H_{[3,1]} & \overline{\overline{w}}^H_{[3,2]} & \overline{\overline{w}}^H_{[3,3]} \end{pmatrix} \tag{3.15}$$

Processor (i, j) of the torus simply gets sub-matrix $\overline{\overline{w}}^H_{[i,j]}$. The activity and delta matrices are divided among the processors in a similar way.

The details of the matrix multiplication algorithm by Fox et.al. will not be discussed. Further information can be found in both [Petrowski] and [Fox2].

The matrix multiplication parallelization performs best on a torus configuration. However, it is not possible to configure a perfect torus when using transputers. A transputer has only four links. Since one of the transputers in a network uses a link for input/output with the host, an extra transputer has to be inserted in one of the rings (see figure 3.14a).

Theoretically, the torus implementation of the matrix.multiplication ought to be superior to any other configuration. However, due to the reduced communication speed resulting from the inserted extra transputer, Petrowski et.al. report that the algorithm for the mesh configuration (see figure 3.14b) executes with approximately the same efficiency. The extra transputer shown in figure 3.14 is actually not necessary with the mesh. For some reason, not mentioned in their paper, Petrowski et.al. use it anyway.

3.3.3 Results and Comparisons

Petrowski et.al. simulate networks of different sizes. They achieve the best results with a net of 64 input, 64 hidden, and 64 output units using a batch size of 64, i.e. all matrices used in the multiplications are square 64 by 64 matrices. Petrowski et.al. use 16 transputers configured as a square torus plus an additional transputer as explained earlier. They report a speed-up of 13.6 out of a possible 16 which is equivalent to an efficiency of 85%. However, as 17 transputers are used, an efficiency of 80% is a more correct figure. Still, this is a higher efficiency than we have been able to attain (using comparable network and batch sizes) with the two data partitioning parallelizations described in sections 3.1 and 3.2.

Hence, we wanted to see if we were able to obtain similar results with the matrix multiplication strategy. Our implementation of back-propagation using the matrix multiplication algorithm by Fox et.al. can be found in appendices B.8 and B.9. We will focus on the torus implementation.

We were not able to achieve the same performance as Petrowski et.al., however. We only got a speed-up of 10.7 equivalent to an efficiency of 67% (or 63% out of 17 transputers). There may be several reasons for this. As mentioned in section 1.6.1 our sequential implementation of back-propagation is between 24% and 38% faster than the implementation of Petrowski et.al. This may be one reason. Another possible reason will be described in the following.

We have made some runs to determine why we got the poor performance. In the first test we removed all communication from our implementation. The speed-up of our own implementation can be found in table 3.3.

Program	Speedup
Parallel 4 by 4 torus	10.7
Parallel without communication	11.8
Parallel with unfolded calculation, with communication	13.3
Parallel with unfolded calculation, no communication	14.6

Table 3.3: Simulation of a 64-64-64 net with 64 patterns in the batch

The communication overhead of our implementation only amounts to 9%. This small number cannot explain the poor performance. The explanation must then be found in some software overhead, i.e. extra index calculations that have been introduced in the parallel implementation. Generally, it is impossible (or at least difficult) to remove such overhead without introducing other kinds of overhead. However, in this case it is possible to remove some of the overhead without to much effort.

Consider the Occam code in figure 3.15. This is part of the code for calculating hidden unit activity values. The `SIZE` variable is the size of the rows/columns of the processor torus, i.e. in a Q by Q torus the value is Q. As can be seen in the figure, the `hidden.weight` and `input.activity` are 3-dimensional arrays. Such arrays are apparently quite time consuming to use.

```
[SIZE][INPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 input.activity:
[SIZE][HIDDEN.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64 hidden.weight:
[HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 hidden.net:

SEQ d = 0 FOR SIZE
  SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
    SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
      SEQ k = 0 FOR INPUT.PR.PROCESSOR
        hidden.net[i][j] := hidden.net[i][j] +
            (hidden.weight[d][i][k] * input.activity[d][k][j])
```

Figure 3.15: Part of hidden unit activity calculation

It is of course possible to rewrite this generally applicable piece of code. If one always uses a 4 by 4 torus of transputers (as Petrowski et.al. do), then the `SIZE` variable is always 4 and the `SEQ d = 0 FOR SIZE` loop of figure 3.15 can be unfolded: The `hidden.weight` and `input.activity` arrays are each placed in four 2-dimensional arrays, thus reducing the amount of index calculations. When similar code segments of our implementation also are unfolded we see a considerable reduction in execution time (see table 3.3). The speed-up of 13.3 is now comparable to the 13.6 obtained by Petrowski et.al. As we only get a speed-up of 14.6 (unfolded an no communication) there is still some software overhead.

The trick that we have just described is of course not generally practicable. If we want an implementation that can be used with any possible number of processors it is not possible to do this. For each new application, essential parts of the program has to be rewritten in order to utilize a specific configuration of processors. However, it shows that when using a specific system it can indeed be worth while to try to optimize the implementation.

3.3.4 Varying the Number of Processors

As mentioned in section 3.3.2 the number of processors used in a simulation has to be quadratic. Hence, the number of points in the speed-up and efficiency graphs of figure 3.16 is very limited. We have used batch sizes of 15, 30, and 60. There is no gain in efficiency in using larger batch sizes than 60 (for the processor numbers in the graph).

This algorithm reaches maximum speed-up for relatively small batch sizes. The two algorithms described in sections 3.1 and 3.2 need larger batch sizes than this algorithm to obtain the same speed-up. However, those two

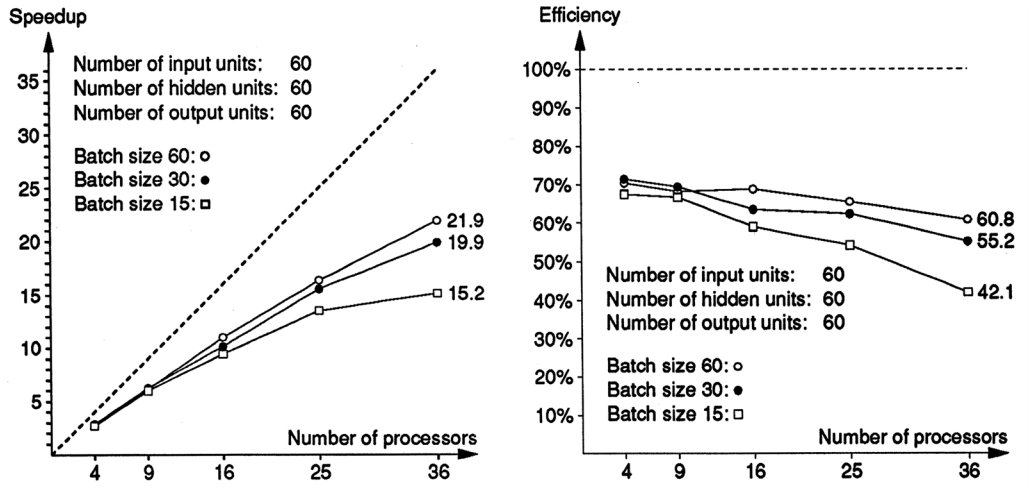

Figure 3.16: Speedup and efficiency graphs when varying the number of processors

algorithms can obtain a better speed-up when much larger batch sizes are used. It seems that this algorithm is useful when small batch sizes are needed.

3.3.5 Varying the Batch and Net Sizes

The efficiency of the parallel matrix multiplication algorithm depends on the size of the dimensions of the matrices involved. The larger the size of the dimensions, the higher efficiency. There are two ways of increasing the size of the dimensions of the involved matrices: One can either simulate a larger neural net or increase the batch size. For a given task, i.e. a fixed net size, there is only one way of increasing the efficiency then: Increase the batch size. This way of increasing the efficiency is very similar to what we have seen in the previous sections.

The graphs of figure 3.17 show the effect of varying the batch size as well as the net size. In this algorithm, the effect of varying these two parameters has almost the same effect. The reason for this is the following: The batch size is the size of one dimension in the activity and delta matrices. The number of input units determines the size of one dimension in the input activity and input to hidden unit weight matrices. Likewise for the number of hidden and output units. Hence, it is not surprising that the two graphs of figure 3.17 are almost identical.

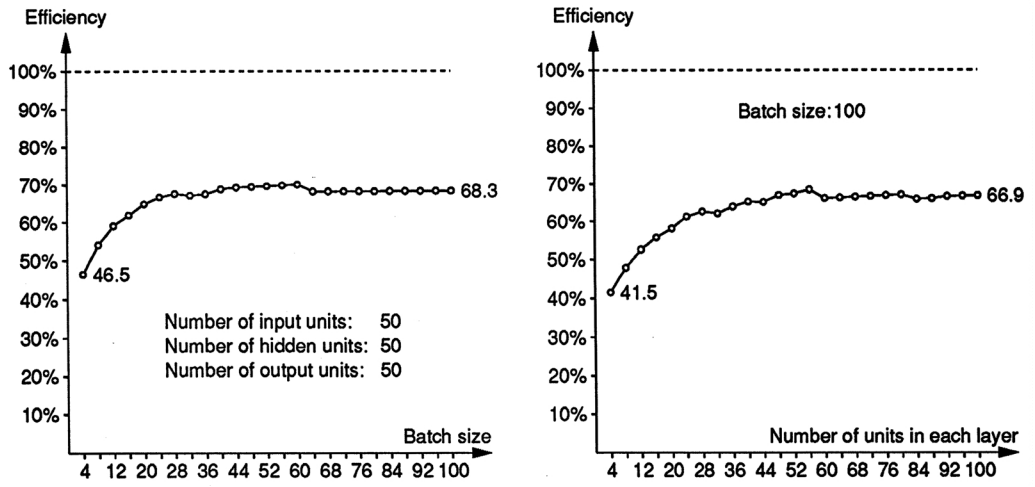

Figure 3.17: Efficiency of 16 processors when varying the batch and net sizes

Neither the choice of batch size nor the number of units in the layers have dramatic effects on the efficiency of the algorithm. If the batch size or the number of units is chosen above 30 the efficiency is almost constant. With respect to the batch size, this is an improvement on the algorithms described in sections 3.1 and 3.2.

3.3.6 Conclusion

The implementation of the matrix multiplication strategy by Petrowski et.al. seemed to yield better results than what we had been able to achieve in the previous two sections. However, the improved performance was probably accomplished through the use of some clever optimization of the implementation which we have not used with the other algorithms. Such clever optimizations are always possible, but generally they are not very interesting.

There are several weaknesses with this third strategy. The matrix multiplication algorithm can only run when the torus or mesh configurations are squares (equally many processors, Q, in the rows and columns), Hence, it is far from possible to use an arbitrary number of processors. Second, in order to attain a high efficiency, both the number of neurons in all layers as well as the batch size have to be divisible by Q.

Chapter 4

Back-Propagation Using Net Partitioning

This chapter is devoted to the development and analysis of an algorithm suited for the pattern updating scheme. This algorithm will include a partitioning of the net among the individual processors.

4.1 Constructing the Parallel Algorithm

In this section we will describe the basis of the parallelization, how the net is divided between the processors, the processor topology, and finally how the actual parallelization of both the forward and backward passes is done. As earlier we will limit our discussion to three-layer networks, that is an input layer, only one hidden layer, and an output layer.

4.1.1 Dividing the Net

When the units and weights of a net are to be distributed among a number of processors, there are not many different ways of doing this. One can slice the net horizontally in three slices such that a single processor contains the units of a particular layer. This is not a good idea because normally there are very few layers in a net. Thus, it is not possible to use many processors. Furthermore, if one insists on using a strict pattern updating scheme this form of parallelization is not possible at all, since the forward and backward passes are really sequential – only the units of one layer at a time make

calculations.

The net can then be sliced vertically such that the units of a layer are distributed equally among the processors. Each processor then contains units from all layers. There are two obvious ways of distributing the weights:

1. All weights feeding into a unit are stored in the processor simulating that unit.

2. All weights leaving a unit are stored in the processor simulating that unit.

There is no real difference between the two strategies. If the first is chosen the forward pass will be easy to parallelize while the backward pass is hard. If the second strategy is chosen it will be the other way round. This will be clear when we go through our parallelization. We have chosen the first strategy.

All units are distributed equally among the processors such that all processors contain the same number of input, hidden, and output units, although the number of processors does not have to divide the number of units in a layer. If the number of processors does not divide the number of units it simply means that some of the processors will simulate one more unit than others. In the following, however, we will assume that everything divides nicely.

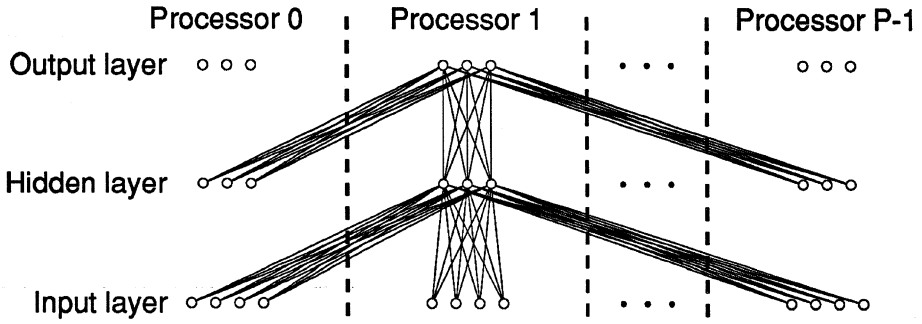

Figure 4.1: The distribution of neural units and weights

The weights of the network are distributed in the following way: All weights feeding into an output or hidden unit are stored in the processor simulating that unit. Naturally, this leads to an even partitioning of the

weights. This is illustrated in figure 4.1 where the weights contained by processor 1 are shown.

The process oriented version of back-propagation discussed in chapter 1 is parallel by nature. As such it would be easy to parallelize with the strategy for unit distribution that we have chosen. Unit processes situated on the same processor would still communicate as in the program of chapter 1. However, there would be a huge number of links between unit processes situated on separate processors. These links could be multiplexed[1] to the limited number of links of the transputers, but we fear that it would be very inefficient. In chapter 1 we saw that the process oriented program was three times slower than the non-process oriented program. With multiplexing on some topology of transputers it can only become worse. We have not investigated the idea of parallelizing the process oriented version further.

The net partitioning strategy for parallelization has been investigated by a few researchers in the field. See [Ernoult], [Zhang], and [Millán] for similar approaches to net partitioning.

4.1.2 The Processor Topology

The simplest possible way to connect a number of processors is with a ring. A ring is easily extended to include more elements and a ring can consist of any number of processors. Also, the ring is a natural topology when one considers the weight partitioning of figure 4.1. Later we show that our use of the ring is optimal, i.e. no time is wasted during communications in the ring. Hence, we have not tried other topologies.

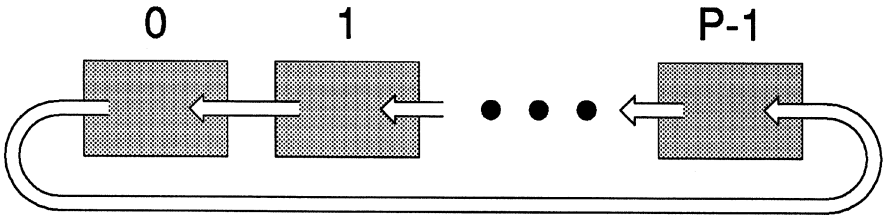

Figure 4.2: Ring of processors

The processors are connected in a ring as illustrated in figure 4.2. The boxes represent processors and the numerals above the boxes are the proces-

[1] See [Welch] for a discussion of multiplexing.

sors' identifications which are needed in the computation. Such a numeral will be referred to as the *processor number*. P is the number of processors in the ring. Note the direction of the links. A processor sends data to its predecessor in the numbering and receives data from its successor in the numbering.

Even though all processors only simulate one part of the input and hidden units, t,hey all have to have the capacity to store the activity of all input and hidden units. This is done, because the activity of all input and hidden units is necessary both in the forward pass and the backward pass of the back-propagation algorithm. By storing all the activities we avoid communicating them again.

Contrary to the data partitioning parallelizations described in the previous chapter, all processors run the same program in this parallelization. The administrator process placed on the root processor differs only from the slaves when it comes to external input/output operations.

4.1.3 Notation

Before we begin the discussion of the parallelization it is necessary to introduce some not ation.

First some convenient abbreviations. As earlier we define P to be the number of processors, NI, NH, and NO to be the number of input, hidden, and output units, respectively, in the network. Furthermore we define PI, PH, and PO to be the number of input, hidden, and output units simulated by each processor. We let $\mathcal{O}$ be the set of output unit indices $(0, 1, \ldots, NO-1)$. The sub-sets of indices for output units simulated by the individual processors are then defined as:

$$\mathcal{O} = \{ \underbrace{0,1,\ldots,PO-1}_{\mathcal{O}_{[0]}}, \underbrace{PO,\ldots,2\cdot PO-1}_{\mathcal{O}_{[1]}}, \ldots, \underbrace{(PO-1)\cdot PO,\ldots,P\cdot PO-1}_{\mathcal{O}_{[P-1]}} \} \quad (4.1)$$

$$\text{Processor } 0 \qquad \text{Processor } 1 \qquad \text{Processor } P-1$$

The hidden and input unit indices are defined in a similar way.

By a_j^H we denote the activity of the j'th hidden unit. The vector of activities of all hidden units is denoted by $\overline{a}^H$, and $\overline{a}_{[p]}^H$ is the sub-vector of

hidden unit activities stored in processor p. This can be written as follows:

$$\overline{a}^H = (\underbrace{a_0^H,a_1^H,\ldots,a_{PH-1}^H}_{\overline{a}_{[0]}^H},\underbrace{a_{PH}^H,\ldots,a_{2\cdot PH-1}^H}_{\overline{a}_{[1]}^H},\cdots,\underbrace{a_{(P-1)\cdot PH}^H,\ldots,a_{P\cdot PH-1}^H}_{\overline{a}_{[P-1]}^H}) \quad (4.2)$$

$$\text{Processor } 0 \qquad \text{Processor } 1 \qquad \text{Processor } P-1$$

The vectors of input and output unit activities, $\overline{a}^I$ and $\overline{a}^O$, are defined in a similar way. Likewise for the vectors of hidden and output unit delta values, $\overline{\delta}^H$ and $\overline{\delta}^O$. Finally, the net input to a unit in the hidden or output layer will be denoted by net_j^H or net_k^O respectively.

Notice, the symbols for net input, activity, and delta do not have a p subscript (for pattern). This is unnecessary as the algorithm we are about to describe conforms to the stochastic gradient method. Patterns are presented one at a time. All processors work on the same pattern.

The weights feeding into the output layer from hidden unit j to output unit k will be denoted by $w_{j\to k}^O$. By $\overline{w}_{\to k}^O$ we denote the vector of weights feeding into output unit k from all hidden units, and $\overline{w}_{j\to}^O$ is the vector of weights feeding into all output units from hidden unit j. As with activities we will denote the sub-vector of weights feeding into output unit k from the hidden units simulated by processor p by $\overline{w}_{[p]\to k}^O$, and $\overline{w}_{j\to[p]}^O$ denotes the sub-vector of weights feeding into the output units simulated by processor p from hidden unit j. The weights from the input layer to the hidden layer are expressed in a similar way.

4.1.4 The Parallelization

The parallelization consists of seven steps, two steps for the forward pass and five steps for the backward pass.

4.1.4.1 Step One

The first step is the calculation of hidden unit activities. This actually consists of two sub-steps, the first being the distribution of input unit activities so that all processors have access to the activity of all input units, and the second being the actual calculation of hidden unit activities.

The entire vector of input unit activities, $\overline{a}^I$, is distributed such that processor p has the sub-vector $\overline{a}_{[p]}^I$. To broadcast all sub-vectors to all processors is simple in a ring, and we do the following: Each processor sends

its own sub-vector to the predecessor in the ring. Then all processors concurrently receive and send sub-vectors calculated by other processors until they all have the entire vector of input unit activities. This takes $P(P-1)$ communications but all P processors communicate in parallel, so the total time for the broadcast is the time it takes to make $P-1$ communications. The broadcast is illustrated in figure 4.3.

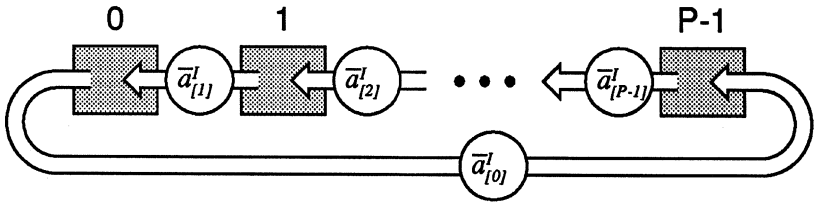

Figure 4.3: Broadcast of hidden unit activities

When the broadcast has finished, all processors have the entire vector of input unit activities, $\overline{a}^I$. Processor p simulates the hidden units with indices $\mathcal{H}_{[p]}$ and all weights feeding into these units are contained by processor p. Processor p can now calculate the net input to all the hidden units it simulates:

$$net_j^H = \overline{a}^I * \overline{w}^H_{\rightarrow j} = \sum_{i=0}^{NI-1} a_i^I w_{i\rightarrow j}^H, \quad \forall j \in \mathcal{H}_{[p]} \tag{4.3}$$

where the operator $*$ is vector multiplication.

With the net input given, the activity of the hidden units can be calculated as follows:

$$a_j^H = f(net_j^H), \quad \forall j \in \mathcal{H}_{[p]} \tag{4.4}$$

where f is the nonlinear activation function. The calculations in equations 4.4 and 4.3 can be performed by all processors in parallel with no communication between them, because all the variables in the equations are stored locally in the processors.

The communication part of step one cannot be avoided. However, the transputers are capable of doing computations and communications over the external links in parallel. We exploit this feature by merging the two sub-steps

of step one. If computation is done while communicating the input unit activities and the computation time is larger than the communication time, the massive communication needed in this step will be of no importance.

The calculation of the net input to a hidden unit, as given in equation 4.3, can be rewritten further in the following way:

$$net_j^H = \sum_{i=0}^{NI-1} a_i^I w_{i\to j}^H = \sum_{q=0}^{P-1} \overline{a}_{[q]}^I * \overline{w}_{[q]\to j}^H, \quad \forall j \in \mathcal{H}_{[p]} \tag{4.5}$$

The last expression of the equation is a sum over all processors. Each $\overline{a}_{[q]}^I$ sub-vector in this sum is stored in a separate processor.

The merging of the two sub-steps begins with each processor sending this sub-vector to the preceding processor in the ring as illustrated in figure 4.3. While these sub-vectors are being communicated on the external channels, all processors calculate their local contribution to the hidden units they simnulate. For all hidden units simulated by processor p the following is computed:

$$\overline{a}_{[p]}^I * \overline{w}_{[p]\to j}^H, \quad \forall j \in \mathcal{H}_{[p]} \tag{4.6}$$

When this calculation has finished, the processor receives the sub-vector from the succeeding processor and immediately sends this on to the preceding processor. The sub-vector received in processor p is the input unit activities of processor $p+1$, and now processor p can calculate the terms:

$$\overline{a}_{[p+1]}^I * \overline{w}_{[p+1]\to j}^H, \quad \forall j \in \mathcal{H}_{[p]} \tag{4.7}$$

which are added to the terms calculated in equation 4.6.

This parallel communication and calculation is continued until the calculation of net_j^H for all $j \in \mathcal{H}_{[p]}$ is completed. Then the activation function is applied to obtain the hidden unit activities.

The input unit activities are needed again in step six when the weight changes are calculated, so upon receiving the sub-vectors of input unit activities here in step one, the entire vector of input unit activities is stored for later use.

The OCCAM implementation of step one can be found in appendix B.10 fold number 13.

4.1.4.2 Step Two

The second step is the calculation of output unit activities. This step is very similar to the first step. The calculation of the net input to an output unit can be rewritten in the following way:

$$net_k^O = \sum_{j=0}^{NH-1} a_j^H w_{j\to k}^O = \sum_{q=0}^{P-1} \overline{a}_{[q]}^H * \overline{w}_{[q]\to k}^O, \quad \forall k \in \mathcal{O}_{[p]} \tag{4.8}$$

As in step one, each $\overline{a}_{[q]}^H$ sub-vector in this sum is stored in separate processors. Again a broadcast is needed, this time of the hidden unit activities.

The OCCAM implementation of step two can be found in appendix B.10 fold number 14.

4.1.4.3 Step Three

The third step is the calculation of delta values for the output units and this can be done with local data exclusively. The delta value of an output unit simulated by processor p with index k is calculated as:

$$\delta_k^O = (t_k - a_k^O) a_k^O (1 - a_k^O), \quad \forall k \in \mathcal{O}_{[p]} \tag{4.9}$$

The OCCAM implementation of step three can be found in appendix B.10 fold number 16.

4.1.4.4 Step Four

The fourth step is the calculation of hidden unit delta values. The calculation of a delta value of a hidden unit, δ_j^H, for processor p is performed as:

$$\delta_j^H = a_j^H (1 - a_j^H)(\overline{\delta}^O * \overline{w}_{j\to}^O) = a_j^H (1 - a_j^H) \sum_{k=0}^{NO-1} \delta_k^O w_{j\to k}^O, \quad \forall j \in \mathcal{H}_{[p]} \tag{4.10}$$

Step four resembles step two, because essentially it is a vector product. However, the weights are not stored in the right place here. When processor p calculates a vector product it uses the weights $\overline{w}_{j\to}^O$ for all $j \in \mathcal{H}_{[p]}$. But these weights are not stored in processor p. They are partitioned equally between all processors, and distributing the weights is far too time consuming.

Equation 4.10 can be rewritten in order to get a sum over the processors. This is done as:

$$\delta_j^H = a_j^H(1-a_j^H)\sum_{k=0}^{NO-1}\delta_k^O w_{j\to k}^O = a_j^H(1-a_j^H)\sum_{q=0}^{P-1}\overline{\delta}_{[q]}^O * \overline{w}_{j\to[q]}^O, \quad \forall j \in \mathcal{H}_{[p]} \tag{4.11}$$

The last expression is thus a sum over all processors. Each term in this sum can only be calculated by the processor storing the weights and delta values. Hence, if $j \in \mathcal{H}_{[p]}$ is the index of a hidden unit simulated by processor a, processor b is able to calculate the terms $\overline{\delta}_{[b]}^O * \overline{w}_{j\to[b]}^O$ needed by processor a in its calculation of $\delta_{[a]}^H$. By $a_{[b]}$ we denote all these terms:

$$a_{[b]} = \{\overline{\delta}_{[b]}^O * \overline{w}_{j\to[b]}^O\}_{\forall j\in\mathcal{H}_{[a]}} \tag{4.12}$$

Thus, $a_{[b]}$ is a *package* calculated by processor b, needed by processor a.

All processors need a package from all other processors. There are a total of $P\ (P-1)$ packages. All these packages are different, so there is no easy way to communicate them between the processors, i.e. it can not be done in exactly the same way as in steps one and two.

There is a way, though, to allow the communications to be carried out in a similar way to step two, We utilize the fact that the packages are not totally independent of each other, they are all part of some summation. In the following we will show that it is possible to communicate and sum the packages in such a way that all communications can be carried out in only $P-1$ sequential steps (with P parallel communications in each step) just as in steps one and two.

We generalize equation 4.12 to:

$$a_{[b,\ldots,c]} = \{\overline{\delta}_{[b]}^O * \overline{w}_{j\to[b]}^O + \cdots + \overline{\delta}_{[c]}^O * \overline{w}_{j\to[c]}^O\}_{\forall j\in\mathcal{H}_{[a]}} \tag{4.13}$$

In this equation a number of packages calculated by the processors from b to c needed by processor a are summed. The sum is a part of the entire sum used in the calculation of $\overline{\delta}_{[a]}^H$.

Figure 4.4 is an illustration of what we do, The numerals above the thick line are the pro- cessor numbers. Between two horizontal lines, some kind of calculation or communication is performed by all processors in parallel. These are the steps, each step is performed in parallel:

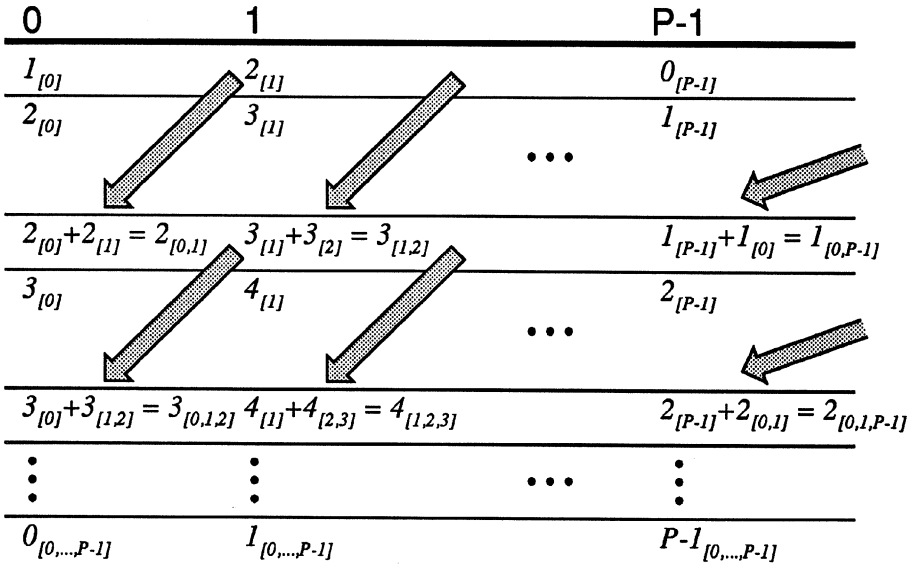

Figure 4.4: Calculation of hidden unit delta values

1. Processor 0 calculates its package to processor 1, denoted by $1_{[0]}$. Processor 1 calculates its package to processor 2, denoted by $2_{[1]}$, and so forth.

2. Both calculations and communications are performed in parallel by the processors. Processor 0 calculates $2_{[0]}$ while sending $1_{[0]}$ to processor $P-1$. Processor 1 calculates $3_{[1]}$ while sending $2_{[1]}$ to processor 0, and so forth.

3. Processor 0 receives $2_{[1]}$ and adds this to the package it just calculated, $2_{[0]}$, and gets $2_{[0,1]}$ which is the package sent next. Processor 1 receives $3_{[2]}$ and adds this to the package it just calculated, $3_{[1]}$, and gets $3_{[1,2]}$. The other processors perform similar tasks.

Now, steps similar to 2 and 3 are repeated until all processors have the entire package needed by the processors themselves, i.e. processor p has the package

$$p_{[0,\dots,P-1]} = \sum_{q=0}^{P-1} \overline{\delta}^{O}_{[q]} * \overline{w}^{O}_{j\rightarrow[q]} \ , \quad \forall j \in \mathcal{H}_{[a]} \tag{4.14}$$

This is the exact sum of the last expression in equation 4.11 for processor p. It is now possible for processor p to calculate $\delta^{H}_{[p]}$.

The OCCAM implementation of step four can be found in appendix B.10 fold numbers 17 and 20. Please note that this step is merged in the appendix with step five and step seven. The code for step five and seven are the internal folds 18, 19, 21, 22, 23, and 24. See section 4.1.6 for an explanation of why the steps are merged.

4.1.4.5 Step Five

The fifth step is the calculation of weight changes of the weights between the hidden and output layer. The equation is:

$$\Delta w^O_{j\to k}(n+1) = \eta\ \delta^O_k a^H_j + \alpha \Delta w^O_{j\to k}(n) \tag{4.15}$$

Because the hidden unit activities calculated in step two were stored, it is possible for the processors to calculate all weight changes with local data exclusively.

The OCCAM implementation of step five can be found in appendix B.10 fold numbers 18, 21, and 23.

4.1.4.6 Step Six

The sixth step is the same as step five, except that the weight changes are for the weights between the input and hidden layer. The equation is:

$$\Delta w^H_{i\to j}(n+1) = \eta\ \delta^H_j a^I_i + \alpha \Delta w^H_{i\to j}(n) \tag{4.16}$$

Again, as the input unit activities calculated in step one were stored, it is possible for the processors to calculate all weight changes with local data exclusively.

The OCCAM implementation of step six can be found in appendix B.10 fold number 25.

4.1.4.7 Step Seven

The seventh step is the actual changing of the weights. Again this can be performed with local data solely, without any kind of communication, and thus in parallel.

The OCCAM implementation of step seven can be found in appendix B.10 fold numbers 19, 22, 24, and 26.

4.1.5 Handling Input and Target Patterns

Depending on the size of the input/target-patterns, there are different strategies for storing these data in the processors. If the entire set of input/target-patterns is small, each patterns can be stored in its entirety in all processors such that all processors have identical copies. This is the case with the NETtalk data set discussed in chapter 5. In this case, step one of the algorithm becomes very simple. As the input patterns (input unit activities) are known to all processors, the processors do not have to broadcast them.

If the set of input/target-patterns is too large to be placed on all processors we distribute each pattern equally among the processors. As mentioned in section 4.1.4, we actually distribute the input units equally among the processors as we do with the hidden and output units, and then let each processor contain exactly the part of every input pattern that is associated with the input units the processor is simulating. Likewise for the target patterns.

4.1.6 Summary

If the communication time of step four is a problem, an obvious improvement of this seven step algorithm is to merge step four and step five, because the calculation of weight changes corresponding to the weights between the hidden and the output layers in step five can be done already when step three is over and thus in parallel with step four. The updating of the weights between the hidden and the output layers (part of step seven) can also be merged with step four.

In section 4.1.2 we claim that a ring of processors is optimal for our purpose. This is easily realized. When the processors communicate in steps one, two, and four, all processors communicate at the same time. Furthermore, and more importantly, no processor communicates any data that is not needed by the processor itself, i.e. no processor is simply passing on data.

4.2 Analysis of the Algorithm

In this section we will run the parallelization on nets of different sizes with varying numbers of processors in order to understand what factors determine the behaviour of the algorithm. The nets are simply nets of convenient sizes; they are not nets for real applications, i.e. no actual learning takes place.

It is only in step one, two, and four of the seven step algorithm communication between the processors takes place. Apart from some overhead which stem from process switching and some extra index calculations, it seems that only the communication in those three steps can prevent a full exploitation of an arbitrary number of processors and thus a maximum speed-up over a sequential version.

As discussed in section 4.1.4 we tried to construct the algorithm so as to perform a minimum amount of communication, preferably in parallel with the calculations. In this section we will analyze to what degree this is achieved and what the determining factors for high efficiency are.

4.2.1 Memory Requirements

One advantage of this algorithm is the low memory requirements. The weights, activities, and delta values of the net are distributed uniformly among the processors. However, the activity values of all input and hidden units are stored on all processors. This is done because these values are used twice. The input unit activities are used both in step one and step six. The hidden unit activities are used both in step two and five.

The memory requirement of each processor (measured in number of `REAL64`s) for variables directly connected with the neural net is given as:

$$NI + NH + PH + 2PO + 2((NI + 1) \cdot PH + (NH + 1) \cdot PO)$$

When the simulated nets are large, the only really important contribution is that of the weights.

4.2.2 The Forward Pass

The two steps of the forward pass are essentially alike, hence we will only analyze one of these steps. As we thoroughly discussed the first step in section 4.1.4 with extensive notation, this is the step we will analyze now. We will show that the calculation time depends linearly on both the number of hidden units simulated by each processor, PH, and the number of input units simulated by each processor, PI. The communication time, however, only depends linearly on PI. In other words, by increasing PH it is possible to increase the calculation time without affecting the communication time. Thus, by choosing PH large enough, the communication time becomes insignificant compared to the calculation time.

When processor p calculates the net input to one of the hidden units it simulates, processor p has to do the following as in equation 4.5:

$$net_j^H = \sum_{q=0}^{P-1} \overline{a}_{[q]}^I * \overline{w}_{[q]\rightarrow j}^H, \quad \forall j \in \mathcal{H}_{[p]} \tag{4.17}$$

This calculation is merged with the broadcast of input unit activities. The relevant part of the OCCAM code for processor p is illustrated in figure 4.5. The sending and receiving of data are executed with high priority. This is done in order to get the link processors initiated as fast as possible. When the link processors take over the communications, the main processor is free to handle the low priority process, the calculation.

```
for all j ∈ H[p] : net_j^H = 0
SEQ i = p FOR NUMBER.OF.PROCESSORS - 1
  SEQ
    q := i REM NUMBER.OF.PROCESSORS
    PRI PAR
      PAR
        send: a_[q]^I
        receive: a_[q+1]^I
      calculate for all j ∈ H[p] : net_j^H = net_j^H + a_[q]^I * w_[q]→j^H
calculate for all j ∈ H[p] : net_j^H = net_j^H + a_[q+1]^I * w_[q+1]→j^H
calculate activities
```

Figure 4.5: The calculation of hidden unit activities for processor p

In each iteration the calculation of $\overline{a}_{[q]}^I * \overline{w}_{[q]\rightarrow j}^H$ term uses PI multiplications and PI additions. Hence, the calculation time in each iteration depends linearly on PI. Since the calculation is done for all hidden units simulated by each processor, i.e. for all $j \in \mathcal{H}_{[p]}$, the calculation time depends linearly on PH as well.

We have measured the time taken to communicate $\overline{a}_{[q]}^I$ terms of varying sizes. PI is between 1 and 10. This is shown in figure 4.6. The time is for parallel sending and receiving as illustrated in figure 4.5. The time for calculating a $\overline{a}_{[q]}^I * \overline{w}_{[q]\rightarrow j}^H$ term for a single hidden unit is also given.

It is obvious from the graphs of figure 4.6 that simulating a single hidden unit per processor is insufficient in the forward pass if a maximum efficiency

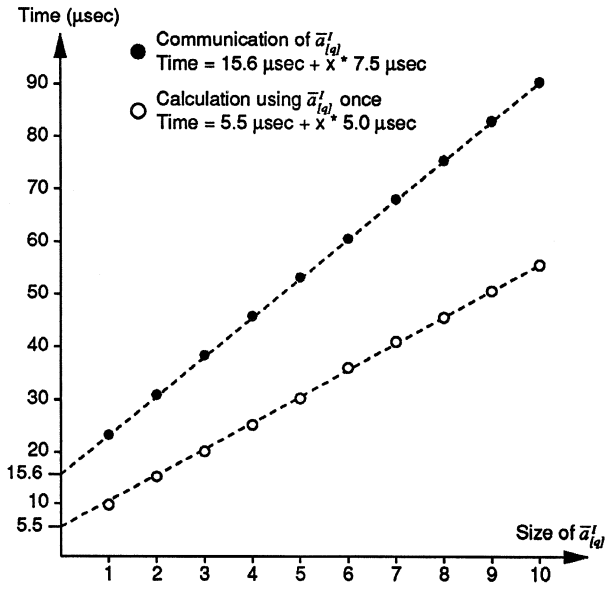

Figure 4.6: Communication and calculation times for $\overline{a}^I_{[q]}$

is expected. When PH is one, the calculation time is smaller than the communication time for any PI. However, raising PH to 2 or 3 will raise the calculation time by the same factor. In effect the calculation time will be larger than the communication time for all values of PI and we can expect a better efficiency.

The merging of calculation and communication in step two of the algorithm is exactly the same as the one just described for step one. For step two the calculation time depends linearly on both PH and PO while the communication time only depends linearly on PH.

The conclusion to the above considerations is that the number of units simulated by the individual processors, i.e. PI, PH, and PO determine how well the simulation of a net is parallelized. If the number of units per processor is large, the calculation time is larger than the communication time in which case we have a good parallelization. If the number of units per processor is small, the calculation is slowed down by communication. This will be demonstrated in section 4.2.4.

4.2.3 The Backward Pass

In step four we also merged the calculations and communications. Figure 4.7 is a part of the OCCAM code for this step.

```
calculate the vector: (p + 1)[p]
SEQ i = (p + 2) FOR NUMBER.OF.PROCESSORS - 1
  SEQ
    q := i REM NUMBER.OF.PROCESSORS
    PRI PAR
      PAR
        send: (q − 1)[p,...,q−2]
        receive: q[p+1,...,q−1]
      calculate the vector: q[p]
    add the two vectors: q[p,...,q−1] = q[p] + q[p+1,...,q−1]
calculate hidden unit delta values
```

Figure 4.7: The calculation of hidden unit delta values for processor p

The communicated packages are vectors of size PH. The calculation of package $q_{[p]}$ as given in equation 4.12 is:

$$q_{[p]} = \{\overline{\delta}^{O}_{[p]} * \overline{w}^{O}_{j \to [p]}\}_{\forall j \in \mathcal{H}_{[q]}} \tag{4.18}$$

The calculation time of a $\overline{\delta}^{O}_{[p]} * \overline{w}^{O}_{j \to [p]}$ term depends linearly on PO. There is one such term for each hidden unit simulated by a processor. Thus, the calculation time of the entire array depends linearly on both PO and PH.

As with steps one and two of our algorithm we see that the calculation time depends linearly on two factors, the number of hidden units simulated by each processor and the number of output units simulated by each processor. Again, the communication time only depends linearly on one of these factors, this time the number of output units simulated by each processor. Similarly, we anticipate a better performance of the algorithm when the number of units simulated by a processor is large.

4.2.4 Effect of Varying the Net Size

In all our executions we have merged the fourth, fifth, and part of the seventh step of the algorithm as suggested in section 4.1.6. Also, we do not take advantage of the on-chip memory of the transputers.

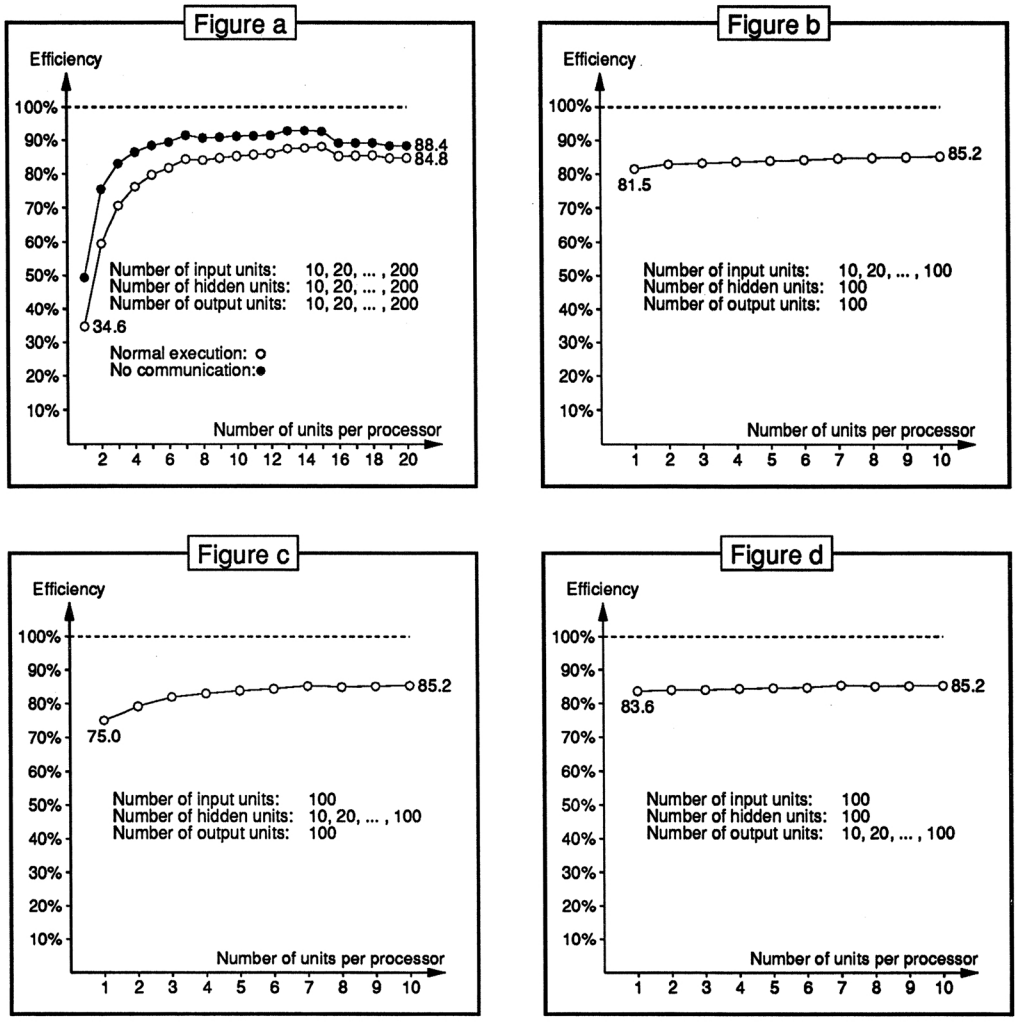

Figure 4.8: Efficiency of 10 processors when varying the net size

The graphs of figure 4.8 are efficiency graphs using a 10 processor simulator. Both the sequential and parallel simulators are executed on nets of exactly the same sizes using pattern updating of the weights. In figure 4.8a the nets are of sizes such that the number of units simulated by each of the 10 processors varies from 1 to 20, i.e. the nets contain 10 input, hidden, and output units, 20 input, hidden, and output units, and so forth.

We see that when there is only one unit per processor in each of the three

layers the parallel simulator is not very efficient, only 35%. When there are two units per processor the efficiency is raised to 59%. With around seven units per processor the efficiency stabilizes on a value of approximately 85%. This is apparently the highest efficiency we can hope to attain.

One reason for the inability to attain a higher efficiency is probably the following: In figure 4.5 the algorithm accesses the elements of the vector $\overline{a}^I_{[q]}$ concurrently in the `PRI PAR` statement. A link processor accesses the elements when communicating the vector, and the main processor accesses the elements when calculating a contribution to the net inputs. Since the link processors and the main processor of a transputer are incapable of concurrent read operations from the same address, the algorithm is slightly slowed down at this stage. Alternatively, a copy could be made of the vector, thus avoiding the concurrent read operations. However, this also takes time and turns out to be even slower.

Another reason for the non-optimal efficiency is illustrated in figure 4.8a. The filled black circles represent runs where the communications have been removed (replaced by `SKIP` statements). The space between the two curves is the communication overhead. The space between the curve with filled circles and the dashed 100% efficiency line is then software overhead of some kind. However, it is not due to incapability of concurrent read operations, as no communication takes place.

We have also measured the importance of parallel calculation and communication. This is done by explicitly performing the calculation and communication parts of the implementation sequentially. We should of course expect longer execution times. Surprisingly, we get the same execution times as with parallel calculation and communication. Hence, it seems that the calculation and communication parts are already performed sequentially in the original implementation. The reason for this is a bit subtle. The transputers are indeed capable of performing calculation and communication in parallel. We have seen this in section 3.2 as well as in appendix A.4. Apparently, the problem lies with the Occam compiler. Naturally, parallel processes are not allowed to change the same variable. The compiler checks this by performing a *usage checking*. In our implementation we have two processes accessing separate parts of the same array. The current version of the compiler is unable to detect that it really is separate parts of the array and gives an error. Thus, for the program to be able to be compiled, we have to switch off the usage checking. Apparently, the compiler generates code such that the two processes are executed sequentially. See [Inmos1, section 5.11] for further

information.

It is possible to get round the problem by implementing the critical parts of the algorithm in machine code. However, that is beyond the scope of this thesis. Secondly, the gain by doing this is minimal. We can at most hope to obtain the efficiency given by the filled circles in figure 4.8a, i.e. the efficiency of the runs where no communication takes place at all.

In the other three graphs of figure 4.8 we lock the number of units per layer in two of the layers while varying the number of units per layer in the third layer. In figure 4.8b we set the number of hidden and output units to 100 and vary the number of input units from 10 to 100 in steps of 10, i.e. the 10 processor simulator has 10 units per processor in the hidden and output layers and from 1 to 10 units per processor in the input layer. By doing this we are able to see the effect of too few input units per processor. However, there is not any noticeable effect. The efficiency is a bit smaller when there is only one input unit per processor.

In figures 4.8c and 4.8d we vary the number of hidden and output units, respectively. We see that when the number of hidden units per processor is too small the efficiency drops a little but not much. The number of output units per processor has no effect on the efficiency as long as enough units are simulated in the other layers.

The conclusion must be: If the number of units per processor is small in all the layers the efficiency is low, but if the processors are short of units in only one of the layers it has almost no effect on the efficiency.

4.2.5 Effect of Varying the Number of Processors

The graphs in figure 4.9 are speed-up graphs for different net sizes with the use of up to 40 processors. In the individual sub-figures the size of the net is unchanged as the number of processors is varied. The sizes of the nets are chosen such that the maximum number of processors (40) divides the number of units in each layer.

In figure 4.9a the net has 40 input, hidden, and output units. A high speed-up is observed with the use of 2 to 8 processors. With the use of 8 processors, all processors simulate exactly 5 units each in the three layers. With the use of 9 processors, 7 of the processors simulate 4 units in the layers but 2 processors still simulate 5 units. The 7 processors are simply idle while the 2 processors do the extra work. There is no gain in using 9 processors instead of 8 because some of the processors still simulate 5 units. Actually,

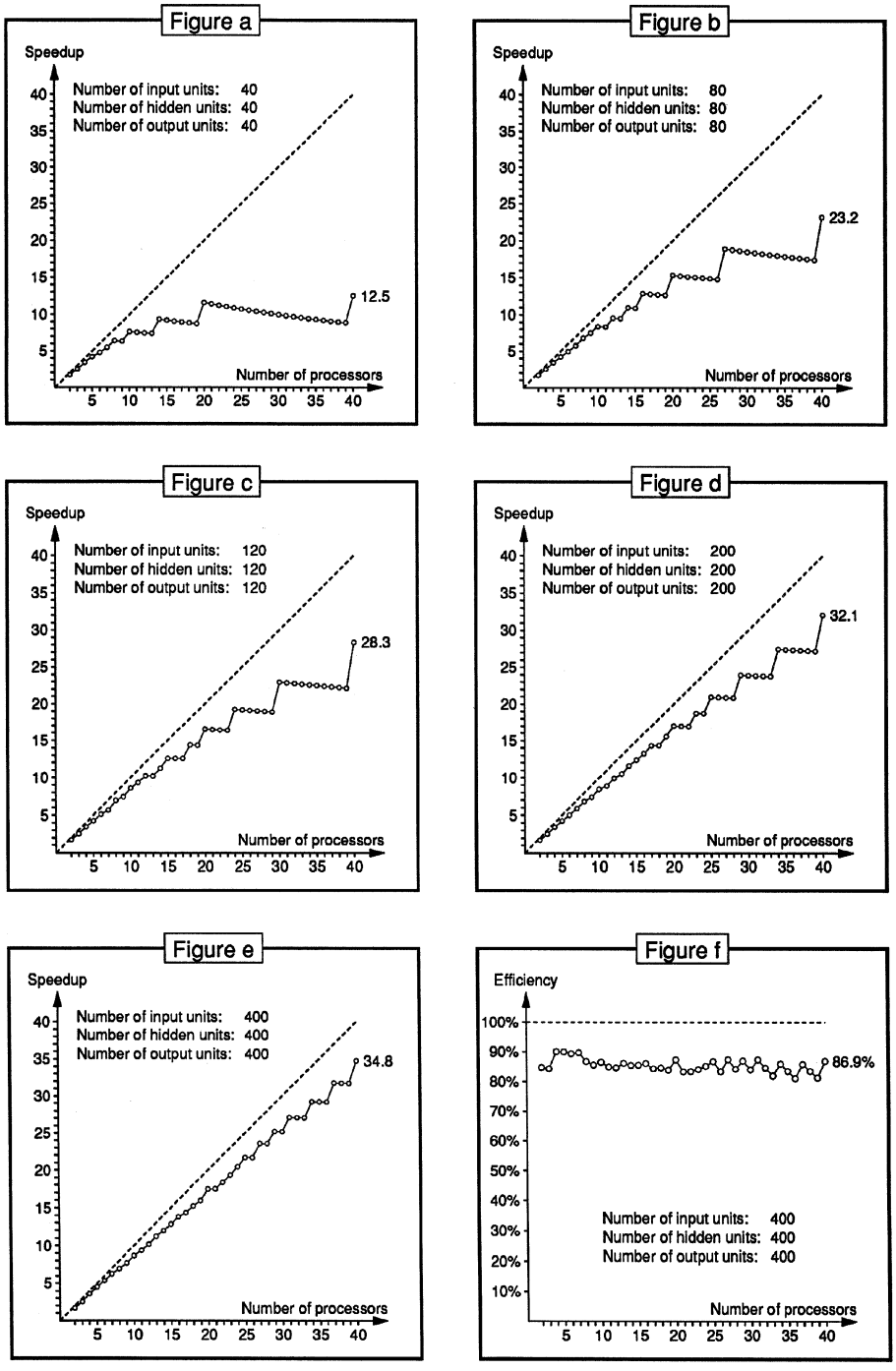

Figure 4.9: Speed-up graphs

using 9 processors is slower than using 8 but only just noticeable. This is due to the extra processor which makes the communication parts of the algorithm take a little longer time. The reason for this is that all processors perform an extra communication.

With the use of 10 up to 13 processors the largest number of units per processor is 4, hence a similar decrease in speed-up as with the use of 8 and 9 processors is observed in the graph. With the use of 14 to 19 processors the largest number of units per processor is 3. With the use of 20 to 39 processors the largest number of units per processor is 2. Finally with the use of 40 processors, they all simulate exactly 1 unit in the layers.

In the other graphs of figure 4.9 we see the same pattern, except that the number of units simulated by each processor is larger, hence the speed-up graphs are better. In figure 4.9a the largest speed-up is 12.5. The best speed-up is of course achieved in figure 4.9e where it is 34.8. Figure 4.9f is the efficiency graph equivalent to figure 4.9e.

The speed-up in figure 4.9a for 40 processors simulating exactly 1 unit in the layers each is only 12.5 which is equivalent to an efficiency of 31%. In figure 4.8a we saw an efficiency of 35% with the use of 10 processors simulating exactly 1 unit in the layers. In figure 4.9b the efficiency of 40 processors simulating exactly 2 units in each of the layers is 58%. The comparable value for 10 processors in figure 4.8a is 59%. We see a similar pattern of correspondence for the other graphs of figure 4.9. Thus, it seems that the number of units per processor in the layers is the determining factor of the efficiency, not the number of processors used. Hence, it should be possible to utilize e.g. twice as many processors with the same efficiency if the number of units in the layers is doubled. This is almost true as we will demonstrate in section 4.2.6.

Note that for the net of figure 4.9e with 400 units in all layers, it is not possible to run the program for 1, 2, 3, or 4 processors on a real application. This is due to the extreme demands of memory used to store the weights. A net with 400 units in all layers uses around 5 Mbyte of memory. The points in the speed-up graph for these processor numbers are based on a modified program which reuses weights in order to save memory. Of course, this modified program is not able to learn any task, it was simply constructed to get the execution times in order to be able to calculate the points of the graph.

However, this shows the advantage of distributing the weights among the processors. With the net of figure 4.9e it is not possible to use a sequential

program, unless the processor used is equipped with more memory or extensive memory swapping on a disk is used. It is possible to run simulations of very large nets when the number of processors is large.

4.2.6 Effect of Scaling the Net with the Number of Processors

In this section we will determine to what degree it is possible to utilize more processors as the size of the net is increased. Figure 4.10 shows efficiency graphs with the use of 2 to 40 processors. The graph with the smallest efficiency is produced in the following way: For each number of processors the size of the net is chosen such that each processor simulates one unit in each layer, i.e. when P processors are used a P-P-P net is simulated. The graph with the second smallest efficiency is the result of simulations of $2P$-$2P$-$2P$ nets with the use of P processors. And so forth with 3, 5, and 10 units per processor.

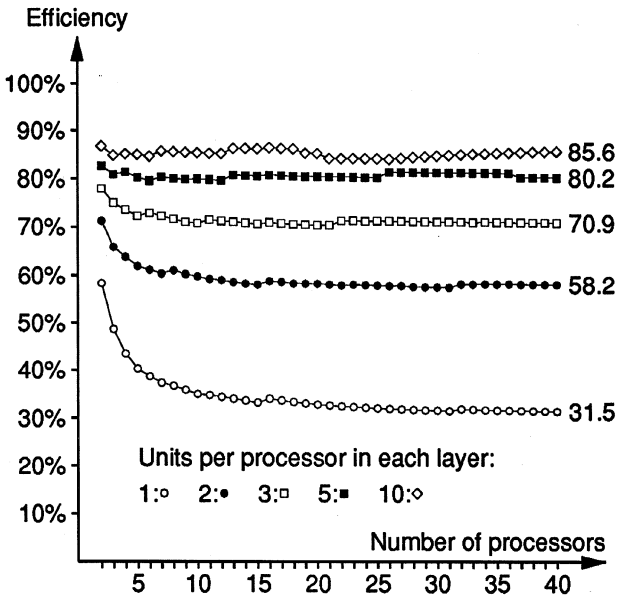

Figure 4.10: Efficiency of scaling the net size with the number of processors

The graph with the smallest efficiency decreases considerably in the begin-

ning. The reason for this is the following: When a small number of processors, P, simulate a P-P-P net, the net is evidently not very large. We have seen earlier that for small nets, the time to calculate the activation function amounts to a considerable part of the entire execution time. This part of the algorithm is of course very well parallelized, hence the larger efficiency in the beginning of the graphs.

As we saw in section 4.2.4 the simulations of nets with few units per processor result in low efficiencies. With the use of at least 5 units per processor the efficiency is almost constant. Hence, it is possible to increase the number of processors without loosing efficiency if the number of units per processor is held constant (thus increasing the total number of units in proportion with the number of processors), Notice, the number of weights in a net increases almost quadratically when the number of units in the layers is increased linearly. Hence, the amount of work is also increased almost quadratically.

The curves of figure 4.10 are almost horizontal lines. If we assume that the graphs continue this way for larger numbers of processors, we can determine whether it is possible to use more processors for a given task. If we have a simulation where the processors simulate 10 units each in the layers with an efficiency of 85.6%, it is possible to utilize twice as many processors (5 units per processor) with an efficiency of 80.2%. It is even preferable to use one unit per processor instead of two. The added processors are not utilized very well, however: One can use twice as many processors with an efficiency of 31.5% instead of 58.2%. Thus, the increase in speed-up is very small.

4.2.7 Effect of Varying the Batch Size

Contrary to the data partitioning parallelizations of the previous chapter, the net partitioning parallelization uses pattern updating of the weights. It is of course possible to update the weights less frequently. With a small modification of the program one can use a batch size of arbitrary size. The modification is very simple: Instead of using the gradient to modify the weights in every pattern presentation, the gradients of a batch are simply summed before they are used in updating the weights. The modification has one negative consequence though: The merging of steps four, five, and seven of the algorithm is not possible any more.

Figure 4.11 shows the efficiency of runs with 10 processors when varying

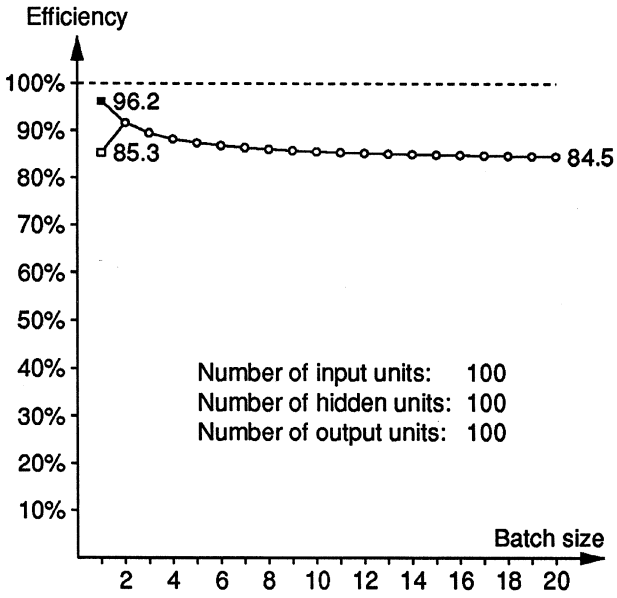

Figure 4.11: Efficiency of 10 processors when using batch updating

the batch size. Notice, for batch size 1, both the modified batch updating (the black square) and the original pattern updating implementation (the white square) is used. For all other batch sizes, the modified implementation is of course used. The modified implementation appears to be faster than the original implementation. This is not so, however. The two parallel implementations are compared to two different sequential implementations. The reason is that the pattern updating sequential implementation also runs faster than the corresponding batch version for batch size 1. The execution times for a 100-100-100 net with 10 learning cycles are given in table 4.1.

Figure 4.11 shows that the efficiency of the batch updating implementation decreases a small amount as larger batch sizes are used. This is because the calculation of weight changes and the actual updating of the weights are performed less frequently. The calculation of those parts are parallelized very well as no communication is done concurrently with the calculation.

	10 processor parallel	Sequential
Pattern updating version	0.49 set	4.2 set
Batch updating version	0.60 set	5.8 set

Table 4.1: Execution times of different pattern updating implementations

4.3 Conclusion

In this chapter a pattern updating parallel back-propagation simulator has been developed. The algorithm uses a partitioning of the network. Furthermore, the communication has been reduced to a level which has little effect on the algorithm.

The algorithm conforms to the pattern updating scheme, i.e. very frequent weight updates can be used without influencing the efficiency of the algorithm. However, a large number of processors can only be applied efficiently when large neural networks are simulated.

These characteristics make this net partitioning parallelization very different from the simple and advanced data partitioning parallelizations discussed in chapter 3. The efficiency of these data partitioning algorithms was almost independent of the size of the simulated network. Furthermore, the batch size was a very determining factor for high efficiency of the data partitioning algorithms.

Chapter 5

NETtalk

In the previous chapters we have analyzed various parallelizations of neural net simulators from a strictly efficiency viewpoint. The data partitioning parallelizations appeared superior to the net partitioning parallelization because good efficiencies were obtained just as long as the batch size was chosen large enough. However, such considerations are not sufficient in neural net contexts. Changing the batch size of the back-propagation algorithm also changes the actual learning process. In this chapter we will show the importance of frequent weight updates, i.e. the bad influence on learning the NETtalk task when a large batch size is used. Hence, the advantage of using a data partitioning parallelization may vanish.

We have chosen to run our tests on a neural net application known as NETtalk. The NETtalk training set will be introduced in section 5.1. In section 5.2 we will describe how NETtalk is implemented on a feed-forward neural net. The performance of NETtalk using various weight updating frequencies will be illustrated in section 5.3. Specific weight updating frequencies (batch sizes) are tested, because these have been tested by other people on different parallel architectures. Finally in section 5.4 we will compare the results obtained by these people with our own results using transputers. The comparison will be a standard execution time comparison as well as a comparison on the ability to learn.

5.1 The NETtalk Data Set

In this section we will describe the NETtalk data set[1] which was constructed by Sejnowski and Rosenberg [Sejnowski] in 1986. Sejnowski and Rosenberg trained a feed-forward neural net to learn the pronunciation of English words. The data set is widely used as a benchmark by researchers in the field of neural nets to test learning speed and generalization ability. By choosing to run some of our tests on this data set, it will be possible to compare our results with results obtained by other people.

The entire data set consists of 20,008 English words with corresponding phoneme and stress information. However, the normal way to use the data is to train on a sub-set of the entire data set only, this sub-set being the 1000 most common English words. The remaining words can then be used to test the generalization ability of the net.

The data set is constructed in such a way that there is a one-to-one correspondence between the letters in a word and the phonemes and stress information. E.g. the word `pronounce` has a corresponding phonemic string of the same length (`prxnW-ns-`) and a string of stress symbols (`>>0>1<<<<`). Actually, the stress symbols consist of both stress and syllable information. There are 52 different phonemes and 5 different stress symbols. See [Sejnowski] for an explanation of the phonemic and stress symbols.

The translation of a single letter in a word to the correct phoneme depends on the surrounding letters in the word, hence Sejnowski and Rosenberg use a "window" of 7 characters. The network is supposed to generate the phoneme corresponding to the letter in the center of the window. The word is scrolled through the window so that all the letters of the word are, one after another, placed in the center of the 7 character window. The neural net is each time presented with all the letters of the window. In this way the network is able to generate all phonemes of a word. This is illustrated in figure 5.1. In the figure the neural net is supposed to generate the phoneme "x" which corresponds to the letter "o" (in the center of the window). To be able to do this, the neural net is presented with all the characters of the entire window, i.e. the letters "_pronou".

The generation of stress information is performed in a similar way. If there are not enough letters to fill the window the space character is used.

With a window size of 7 characters as used by Sejnowski and Rosenberg,

[1] See [Sejnowski] for an explanation on how to obtain this data set.

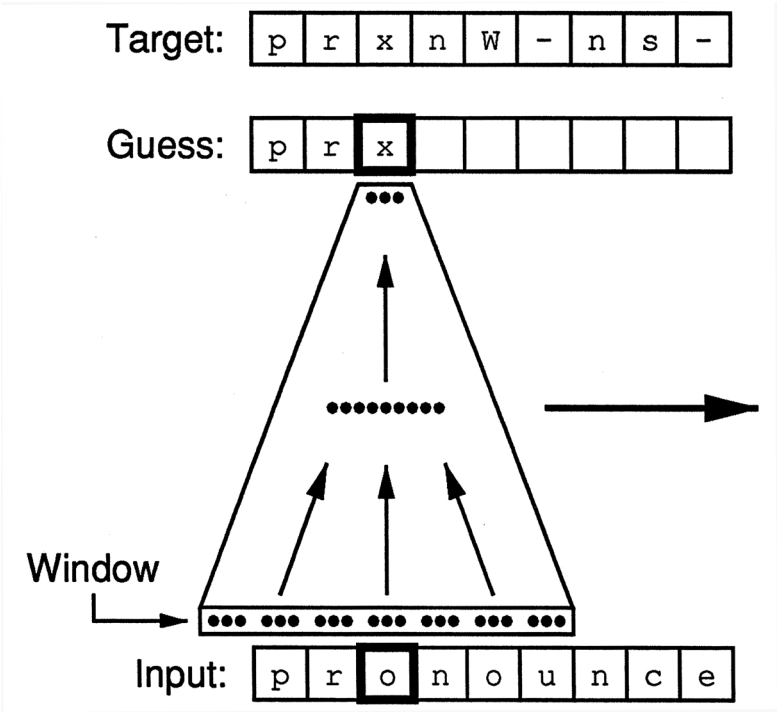

Figure 5.1: The window presentation used by NETtalk

it is obvious that the neural net will not be able to pronounce all words correctly. E.g. the neural net will at most be able to pronounce one of the words *finite* and *infinite* correctly. When the neural net is presented with the string "`finite_`" in the 7 character window in order to determine the phoneme of the letter in center of the window, the network is unable to tell whether the whole word is *finite* or *infinite*. The pronunciation of the letter 'i' (fin i te or infin i te) is different according to which of the two words is presented. The same problem arises with words like *though* and *thought* and other examples can be found. The problem is of course easily solved with a larger window size. However, our goal is not to construct the neural net performing best on the NETtalk data set but to be able to compare our results with results obtained by others. Since a 7 letter window is the window size used by all other people, this is the window size we will use.

5.2 The Neural Network Implementation

An orthogonal representation[2] with respect to each letter is used for the input units, i.e. with 26 different letters in the alphabet and a window size of 7, there are a total of 182 input units. This means that for each letter in the window only one of the corresponding 26 input units is active. When a letter in the window is a space character none of the corresponding 26 input units are active.

Contrary to what is used with the input units, Sejnowski and Rosenberg use a distributed representation[2] for the output units. There are a total of 52 phonemes which are expressed in terms of 21 articulary features, such as point of articulation, voicing, vowel height, and so on. The 5 stress symbols are represented orthogonally, thus giving a total of 26 output units. When comparisons with results of others are to be made, 60 units are normally used in the hidden layer. It is possible, however, to achieve better performances with respect to learning when more hidden units are used. Again, our goal is not to increase the neural net performance, but the ability to compare results.

The NETtalk data set, which consists of words, phonemes, and stress symbols, is very small compared to training data of other neural net applications. The NETtalk data set only takes up 15 Kbyte for the 1000 word data set.

The data requirements of NETtalk are so small that it is possible to place a copy of the entire data set on all transputers. This is an advantage for the net partitioning parallelization. In this way there will only be an initial distribution of data. No data will need to be communicated during the simulation. All processors have direct access to all patterns. As a result, step one of the net partitioning parallelization will be simple as mentioned in section 4.1.5.

Since a binary (and orthogonal) representation is used for the input units, the calculation of hidden unit activities can be simplified. For a single letter in the window only one of the corresponding 26 input units is active (with activity l), the 25 other input units are inactive (with activity 0).

[2]When all pattern vectors are orthogonal, the representation is called *orthogonal*. If the pattern vectors are compressed or coded in some way, the vectors might not be orthogonal. This is called a *distributed* representation.

Hence, in the formula:

$$a_{pj}^{H} = f(net_{pj}^{H}) = f(\sum_{i=0}^{NI-1} a_{pi}^{I} w_{i \rightarrow j}^{H}) \tag{5.1}$$

there is no need to perform all the multiplications. One can simply sum the weights leading from the input units with activity 1. This is always possible to do when a binary representation is used.

Notice, that contrary to the runs of the previous chapters, we do of course utilize the on-chip memory of the transputers in the runs with NETtalk.

5.3 Simulations and Results

In this section we will show how the choice of batch size influences the learning capability, We have used several specific batch sizes because these have been used by others in their parallelizations on various architectures. E.g. Pomerleau et.al. [Pomerleau1] use batch sizes of 90 and 190 on the *Warp* machines.[3] Witbrock et.al. [Witbrock] use a batch size of 512 on the *GF11* machine.[4]

All results presented in this section are obtained with transputer implementations. The results agree with the results obtained by Sejnowski and Rosenberg. For each execution we give both the learning rate as well as the momentum value we have used. In this way the results should be reproducible.

Sejnowski and Rosenberg use two measures of performance. The output is considered a *perfect match* if the value of each output unit is within a range of 0.1 of its target value.

The output is considered a *best guess* if the correct target vector is the vector among all 52 possible target vectors which makes the smallest angle

[3]The main ingredient of the Warp architecture is a linear array of 10 individual processors with distributed memory (MIMD). Each processor has a peak arithmetic performance of 10 Megaflops. The processors can communicate with their neighbours in the linear array at an impressive speed of 80 Mbytes/set (32 times faster than the transputers). The cost of a 10 processor Warp is $350.000. A 20 processor Warp is under construction.

[4]The IBM GF11 is an experimental distributed memory SIMD machine with 566 processors and a total peak arithmetic performance of 11.4 Gigaflops. The processors are connected via a programmable network. The communication speed of this network is not given in the paper. Although the GF11 has 566 processors, Witbrock et.al. only use 512 of these.

with the vector of actual output unit activities. I.e. all 52 possible target vectors (each consisting of 26 values) are *compared* to the generated output vector, and a best matching target vector is found. If this target vector is the correct target vector, the output is a best guess. Consequently, the output vector may be very different from the correct target vector when the best guess criterion is used.

It is obvious that the best guess criterion gives the highest percentages of correct responses (a perfect match is always a best guess but the converse is not always true). However, using the best guess criterion is computationally much more demanding than using the perfect match.

We trained the nets on the 1000 most commonly used English words. After each presentation of 1000 words the nets were tested on the same 1000 words to measure the performance. The graphs of figure 5.2 show the performance of the nets when trained using different weight updating frequencies. In all training sessions the words were presented 50 times each. The words were presented in random order. The momentum was set to the normally used value 0.9.

The graphs are average graphs of several runs with different random starting points in weight space. The difference in performance is hardly noticeable between runs made from different starting points in the weight space. This is not surprising when the relatively large number of weights in the net (which amounts to 12566 including bias weights) is taken into account.

Figures 5.2a and 5.2b are the results of using word updating, i.e. the weights were only updated after all letters of a word had been presented. This is the weight updating frequency used by Sejnowski and Rosenberg. Notice that weights were updated very frequently but not regularly since words differ in length (from 1 to 14 with an average of 5.4 in the set of 1000 words). A learning rate of 0.2 was used.

Figure 5.2a is the performance in percent measured both as best guess and perfect match. We see that the best guess performance quickly rises to around 90% and then only increases slightly after that, reaching 94.5% after 50000 word presentations.[5] The perfect match performance exhibits a more moderate increase. It reaches 64.7% but seems to be able to attain an even higher percentage if the net was trained more.

[5]With 120 hidden units instead of 60, the performance reaches 98% after 50000 word iterations.

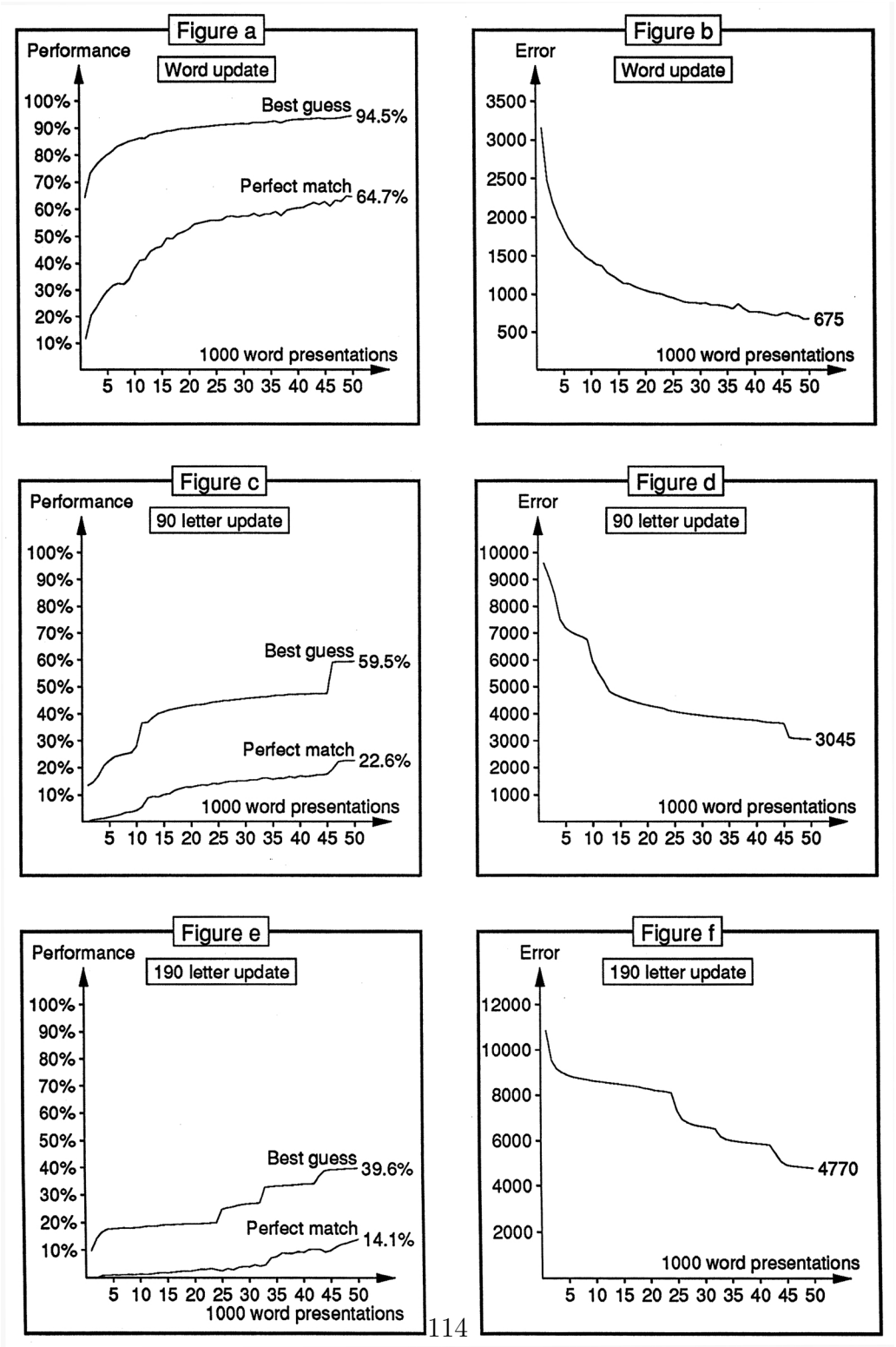

Figure 5.2: Performance of NETtalk after 50000 word presentations

Figure 5.2b shows the corresponding standard measure of error as calculated by:

$$E = \sum_{p=1}^{5438} \sum_{j=1}^{26} (t_{pj} - a_{pj})^2 \tag{5.2}$$

which is the squared sum over the 26 output units and all 5438 patterns (which is the number of letters in the 1000 words). For comparison, the average error of an untrained net is approximately 22600. The nets are tested for the first time after 1000 word presentations. Evidently, the error drops considerably during the first 1000 word presentations.

Figures 5.2c and 5.2d are the results of updating the weights after presenting 90 letters (a batch size of 90). This is the weight updating frequency used by Pomerleau et.al. [Pomerleau1] on the 10 processor Warp architecture. A learning rate of 0.02 was used. Such a small learning rate was necessary in order to prevent *oscillation*: As larger batch sizes are used more component gradients are summed before updating the weights. Hence, a smaller learning rate is necessary if the magnitude of a weight change is to be kept on an acceptable level. The result of too large weight changes is oscillation — no learning at all takes place. See [Rumelhart] for more information on which learning rate to use.

In figure 5.2c we see that the net attains a considerably lower performance measured both as best guess and perfect match than with word updating. Also, the graphs are more irregular. Furthermore, with word updating the error gets as low as 675, whereas the error only drops to 3045 after 50000 word presentations in figure 5.2d.

Figures 5.2e and 5.2f are the results of updating the weights after each presentation of 190 letters. This is the weight updating frequency Pomerleau et.al. [Pomerleau1] is expected to use with the 20 processor Warp architecture under construction (if the batch size per processor is the same as with the 10 processor Warp). A learning rate as small as 0.01 was used, again to prevent oscillation. The figures show an even poorer performance than with the previous two weight updating frequencies. The error only drops to 4770 after 50000 word presentations.

The graphs for both 90 and 190 letter updating seem to suggest that a prolonged training could increase the performance. Hence, we trained the nets four times as long using both weight updating frequencies. The results can be seen in figure 5.3. With the 90 letter weight updating frequency the

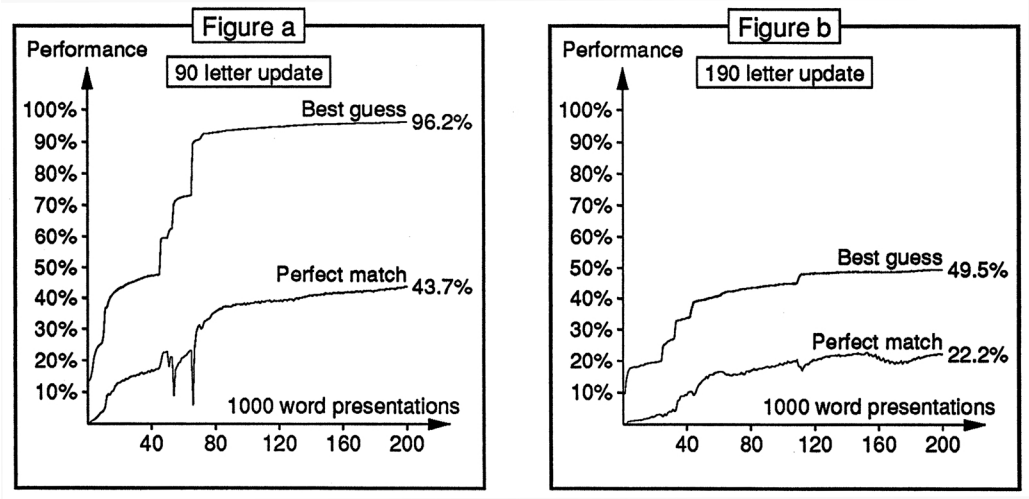

Figure 5.3: Performance of NETtalk after 200000 word presentations

net was indeed able to perform better. With 190 letter updating, however, we did not observe a similar improvement. It might take a huge number of additional presentations to improve the performance.

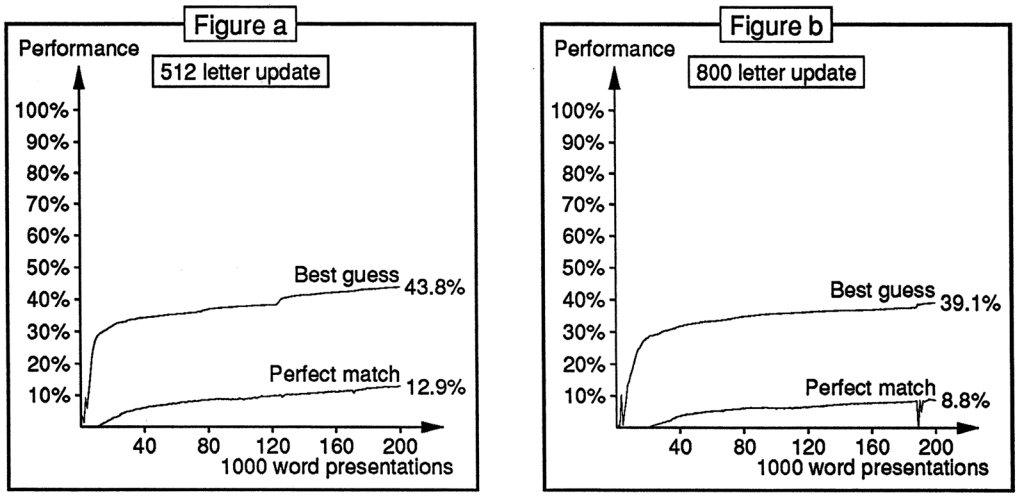

Figure 5.4: Performance of NETtalk after 200000 word presentations

In section 5.4 we are going to compare NETtalk implementations on different parallel architectures. Hence, we will finally give the performances

of two other weight updating frequencies. Figure 5.4a shows the performance when a batch size of 512 was used (the smallest batch size that can be used by Witbrock et.al. with their 512 processor GF11).

Figure 5.4b is the result of an implementation using a batch size as large as 800. Both have been run for 200 thousand word presentations with learning rates of 0.002 and 0.001 respectively.

When the results of figures 5.2, 5.3, and 5.4 are compared it is obvious that the performance deteriorates as the batch size is chosen larger and larger. It is especially the perfect match criterion that gets degraded when large batch sizes are used.

The previous observations lead to one conclusion: As the weights are updated less frequently more pattern presentations are required to obtain the same performance, if at all the same performance can be obtained. This is also a conclusion found by Bourrely [Bourrely] who has parallelized back-propagation on the Hypercube with up to 32 processors using data partitioning. Bourrely's experiments are not performed with the NETtalk task. He trains a net which is supposed to recognize handwritten numbers. Bourrely writes:

> Given that, in practice, the [stochastic gradient method] is the only one which converges quickly, it is easy to understand the decrease in learning speed when the algorithm used gets closer to the [true gradient method]. For high values of [batch size] or [number of processors], the parallelized algorithm diverges or stagnates, This drawback of the parallelization totally eliminates the positive effect on calculation time.

Witbrock et.al. have made similar observations with the 512 processor GF11 architecture using data partitioning with a batch size not less than 512. They write [Witbrock]:

> Since the NETtalk training set is suited to frequent updates, our simulator would probably take more pattern presentations to learn this task than the Warp (with fewer processors) or the Connection Machine simulators (with a different form of parallelism). It is even conceivable that our simulator might take longer to learn NETtalk than one of these other simulators.

Although efficient parallelizations exist using data partitioning (with large batch sizes), they may not be very useful. If the ability to learn deteriorates

severely when a large batch size is used, the effect on learning speed of using many processors may be lost.

Witbrock et.al. seem to suggest that the NETtalk data set is especially suited to frequent weight updates (as the handwritten character recognition task used by Bourrely apparently is), and that this is not necessarily the case for all other neural net problems. I.e. there may exist data sets for other neural net applications for which rare weight updates work just as well as or even better than frequent updates. Vogl et.al. has reported [Vogl] that epoch updating combined with a dynamically changing learning rate and momentum factor may accelerate the convergence rate for some class of moderately complex problems. If this is the case, data partitioning parallelizations may indeed be applied successfully. However, we have not been able to find descriptions of such data sets anywhere else in the literature.

5.4 Comparisons

In this section we will compare implementations of back-propagation on different parallel architectures. Normally, comparisons between different parallelizations are easy to make, because it is the same algorithm that has been used in the different parallelizations. This is the problem with parallelizations of back-propagation (see section 2.5). Some implementations are parallel versions of the batch updating algorithm and exploit this. Some implementations use the pattern updating algorithm. Since the two different strategies for updating weights result in quite different performances (as could be seen in the previous section) comparisons are not so easy. At the risk of jumping to untenable conclusions we will give it a try, anyway.

When the performances of neural net simulators on different architectures are compared, a measure known as *connections per second* (or CPS) is often used. In [Witbrock] CPS is defined as: The number of connections (including bias connections from the imaginary unit) times the number of patterns presented divided by the total running time.

CPS is in no way a perfect measure. The measure does not account for the frequency of weight updates, i.e. whether the net is able to learn anything is irrelevant! Neither does the measure account for the fact that connections from the input layer require less computation. In spite of these reservations, the CPS measure is the only widely recognized measure and is still used. It is obvious that the measure favours batch updating algorithms.

We have made three runs with transputer implementations to obtain the CPS measures. We used a 30 processor net partitioning implementation with pattern updating, and a 40 processor data partitioning implementation (the simple version of section 3.1) using batch sizes of 800 and 4096 (the batch size of 4096 is only included to show that it is possible to obtain a very high CPS count with 40 transputers. We do not recommend the use of such a high batch size). The learning capability of runs with batch size 800 can be seen in figure 5.4 of the previous section. We have not included a figure which shows the learning capability of pattern updating and batch size 4096. The performance of pattern updating is very similar to that of word updating. Batch size 4096 is expected to yield an even worse performance than batch size 800.

The 30 transputers used in the net partitioning implementation are not applied efficiently. The units in each layer are divided equally between the transputers of that algorithm. However, there are only 26 output units. With the use of 30 transputers, 4 of the transputers do not simulate an output unit. The number 30 divides the number of hidden units (60). This is why we have used 30 transputers, no more and no less.

Tables 5.1 and 5.2 show the capabilities of back-propagation implementations on different machines measured in million CPS (MCPS). The numbers are obtained from [Witbrock]. The machines of table 5.1 use implementations of the pattern updating algorithm. The machines of table 5.2 use the batch updating algorithm. Notice, the estimation for the 20 processor Warp in table 5.2 was made by Pomerleau et.al. themselves.

Machine	MCPS
Sun 3/75	0.01
16384 processor CM-1	2.6
30 transputers, net partitioning	4.1
CRAY-2	7

Table 5.1: Performance of different architectures using pattern updating

The CPS count lists are peaked by GF11 with an impressive 180 MCPS and similarly impressive figures for the Warp architectures. However, as quoted from [Witbrock] in the previous section these high figures are not so impressive when the learning capabilities are included in the evaluation. In figure 5.4 we saw the performance when a batch size of 512 was used, which

Machine	MCPS	Batch size
40 transputers, simple data partitioning	5.1	800
40 transputers, simple data partitioning	11.8	4096
10 processor Warp	17	90
20 processor Warp (estimated)	32	190
65536 processor CM-2	38	4096
512 mocessor GF11	180	512

Table 5.2: Performance of different architectures using batch updating

is the smallest batch size that can be used with the GF11 (when using 512 of the available processors).

The learning performances of the Warp architectures with 10 and 20 processors were shown in figures 5.2 and 5.3. When Pomerleau et.al. almost double the CPS count with the use of a 20 processor Warp instead of the 10 processor version, this is not very impressive when the reduced learning is taken into account. In order to exploit the 20 processors fully a batch size of 190 instead of 90 is necessary. The effect on learning is clear from figures 5.2 and 5.3. The figures suggest that at least twice as many word presentations are required in order to obtain the same learning performance. Hence, the effect of using twice as many processors is not that the task is learned twice as fast. On the contrary, it may even take longer time to achieve a certain degree of learning performance with the use of twice as many processors.

The 7 MCPS of a CRAY-2, the 2.6 MCPS of a Connection Machine[6] or the 3.5 MCPS of 30 transputers using net partitioning are more genuine since these figures are for algorithms using pattern updating.

It is possible to make learning time comparisons between e.g. the 10 processor Warp using a batch size of 90 and 30 transputers using word updating. I.e. how much time is used by the two different implementations before a certain performance is obtained. If the graphs of figure 5.2a are compared to those of figure 5.3a, we see that approximately the same performance is achieved with word updating after 50 thousand word presentations and with 90 letter batch updating after 200 thousand word presentations (measured as best guess). In other words: The 10 processor Warp imple-

[6]See [Blelloch] for a description of the 16384 processor Connection Machine and an implementation of NETtalk on this.

mentation uses approximately 4 times as many word presentations to obtain the same performance as the 30 processor transputer implementation. It is tempting to divide the Warp's 17 MCPS performance by 4 and obtain a 4.3 MCPS performance which is then only slightly superior to the performance of the transputers.

Such comparisons are not fair, however, or at least they cannot be generalized to other neural net applications. There may exist applications in which the advantage of using frequent weight updates is not as large as the advantage demonstrated for the NETtalk application. For such applications the performance of the Warp will still be superior to that of the transputers.

5.5 Conclusion

Two different approaches to parallelizing the back-propagation algorithm have been considered. The data partitioning approach of chapter 3 and the net partitioning approach of chapter 4. When large batch sizes arc used the data partitioning parallelizations can use many processors efficiently (almost) independently of the neural network size. The net partitioning parallelization works independently of the batch size (frequent weight updates are possible) but needs a large neural network in order to utilize many processors.

With the NETtalk application we have seen that the data partitioning parallelizations are superior to the net partitioning parallelization — as long as the CPS measure is the only criterion. When other machine architectures are considered, some of these perform a great deal better than the transputer implementations (again when the CPS measure is the only criterion).

As stated earlier, the CPS measure is not very useful. E.g. with 40 transputers we can achieve 11.8 MCPS with a batch size of 4096. The network, however, is totally unable to learn anything within a reasonable time span when the weights are updated this rarely.

The degradation in learning performance when large batch sizes are used has been observed by several researchers. In section 5.3 we cited Witbrock et.al. [Witbrock] and Bourrely [Bourrely] who criticized their own parallelizations. Singer [Singer] compares different Connection Machine implementations of back-propagation. In this comparison he uses expressions as "creative benchmarking" and he writes: *In spite of the incomplete status of the on-going eflorts to provide a standard set of tasks and measurement specifications for ANNs, the pressure to generate simple numerical scores by which*

artificial neural network programs can be judged is great; too great, in fact, to resist.

Pomerleau finds, in [Pomerleau2], that: *The traditional method of measuring simulator performance in terms of raw connections per second is misleading. Specifically, neural network implementations which simulate a large number of patterns in parallel, and which therefore require the simulation of many patterns before performing a weight update, may learn less quickly and less robustly than implementations which update weights more frequently.*

It is only possible to compare different implementations of back-propagation if the exact same parameters are used, i.e. training data, network size, and weight updating frequency. NETtalk is an attempt to make a benchmark task such that this is possible. The NETtalk task is used in many different ways, however. E.g. the normal number of hidden units to be used in the network is 60. Zhang et.al. [Zhang] use 80 hidden units because their parallelization on the Connection Machine works well with this number.

Because the NETtalk data set is used in many different ways, it is difficult to compare the performance of our transputer implementations to the implementations on other architectures. In this chapter we have tried. We believe we have shown that a transputer system is indeed worth considering as the basis of a neural net simulator.

Chapter 6

Conclusion

The subject of this thesis has been the parallelization of the back-propagation neural net learning algorithm.

We have seen that both of the parallelization strategies introduced in chapter 2 allow successful parallel transputer implementations to be developed. The question of which of the parallel neural net simulators should be preferred cannot be answered in general. The choice must depend on the specific neural net problem in concern, and the desired weight updating frequency.

If a large batch of training patterns is preferred (for problem specific reasons) one of the data partitioning parallelizations will probably allow a larger number of processors to be used, and a higher speed-up to be obtained, than the net partitioning parallelization will, at least if the size of the neural net is relatively small. If the number of available processors means that the batch size per processor will be relatively large, the simple implementation of the data partitioning strategy will most likely produce the highest speed-up. However, if the number of available processors is large compared to the size of the batch, then the advanced implementation is expected to give the best results, since it is able to run with a much higher efficiency for small batch sizes per processor.

If frequent weight updates are desired, the net partitioning parallelization should be used, since the efficiency of this algorithm does not depend on the batch size being large. Furthermore, it is the only algorithm capable of using pattern updating. However, for a high efficiency to be obtained each processor must handle several units in each layer of the neural net. This means that only a very limited number of processors can be applied to the

simulation of small nets. But if both the neural net size and the size of the batch are relatively small, simulating one learning cycle will not take up much time even when using the sequential algorithm.

It should also be taken into consideration, when comparing algorithms, that input patterns consisting of binary values allow the simple implementation of the data partitioning strategy and the net partitioning parallelization to be speeded up. The reason for this is, that when input patterns consist of binary values the amount of computations associated with the weights between the input and the hidden layer can be reduced, as described in section 5.2. In the advanced implementation of the data partitioning strategy and in the algorithm using matrix multiplication, such a reduction will not always reduce the work of all processors to the same degree. Thus, the result will likely be that some processors will have to wait for other processors to finish, thereby removing much of the benefit of the reduced amount of computation.

In connection with the choice of parallel neural net simulator it should be noted, that the size of the batch should not be chosen large simply to allow an efficient application of one of the data partitioning parallelizations, since increasing the batch size may severely reduce the learning speed and degrade the quality of the learning in terms of generalization ability. We have described an example of this phenomenon in chapter 5 about the NETtalk application. In this chapter we saw that even though the CPS count of the net partitioning parallelization is much smaller than that of the simple implementation of the data partitioning parallelization, it clearly outperforms the latter algorithm when the vastly improved learning performance of word or pattern updating is taken into account. In fact, on the NETtalk problem our implementation of the net partitioning strategy was seen to be comparable in terms of learning speed to the performance achieved by Pomerleau et.al. on the 10 processor Warp computer [Pomerleau1].

In the NETtalk chapter we stress the importance of this observation, that one cannot evaluate the performance of a parallel neural net simulator just by measuring the number of pattern presentations performed per second, without taking into consideration whether any learning takes place in the net. Only simulators that use exactly the same kind of learning method, including the frequency of the weight updates, can be compared in a fair way by looking at the number of pattern presentations per second, or some similar measurement of speed.

Otherwise, different neural net simulators should be compared by measuring the absolute execution time necessary to achieve a certain level of

perfection on some specific neural net problem. This, however, is not a satisfactory solution, either, since the influence of batch size on learning speed and generalization ability varies from problem to problem. There may exist problems where using epoch updating of the weights is superior to more frequent weight updates in terms of both learning speed and quality. In other words, it is probably not possible to give a general recommendation as to which parallelization should be preferred. Different parallel neural net simulators are simply suited to different kinds of neural net problems.

A point of natural concern is whether the parallelizations described in this thesis are restricted to running the ordinary back-propagation algorithm, or whether it is possible to use some or all of the considerations and techniques described here when parallelizing other neural net learning algorithms. It turns out, that the greater part of the computational work in a conjugate gradient algorithm is associated with the calculation of the gradient of the error function, as in standard back-propagation. This indicates that parallel conjugate gradient algorithms can be constructed as extensions of the algorithms we have described in this thesis. We have in fact performed a number of preliminary experiments, and results not reported here suggest, that the efficiency which can be achieved in these algorithms is comparable to that of the standard back-propagation parallel algorithms.

We believe that we have demonstrated in this thesis that a transputer system is an excellent choice for running neural net simulators. It is competitive in performance and much cheaper than most of the alternatives.

Appendix A

The Transputer

In this appendix we briefly explain the most important aspects of a transputer. See [Inmos1] for more information. We have access to a MEIKO transputer system with 48 transputers located in Odense. This will also be discussed.

A.1 The Transputer Architecture

A transputer is a very complex VLSI-chip containing both a processor, a small amount of memory, and *links* for communication with other transputers. Several different types of transputers exist, the one we are using is the INMOS IMS T800-30. Figure A.1 is a simple diagram showing the main parts of the transputer. Apart from the processor (CPU), memory (RAM), and links, the transputer contains an on-chip floating point unit (FPU). This FPU has a peak performance of 2.25 MFLOFS. The memory interface is for adding external memory, which is necessary since the on-chip memory amounts to only 4 Kbyte. However, the on-chip memory is much faster (approximately 3 times) than any external memory.

There are four links for communication. Thus, a transputer is capable of communicating directly with four other transputers or different kinds of processors respecting the communication protocol. Each of these links consists of a *link-interface* and two wires, one wire for receiving data and one for sending data. A link-interface is an on-chip processor working independently of the main processor. Furthermore, the link-interface has direct access to the memory. The communication speed of the links is 20 Mbit per second.

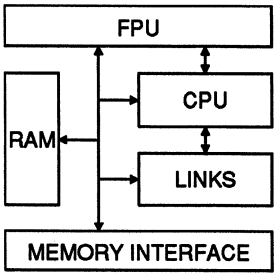

Figure A.1: Simple diagram of an IMS T800

The transputer can handle arbitrarily many processes, the only limitation being the size of available memory. Process switching is very fast, less than 1 μsec on average and not more than 3 μsec in worst case. Processes can have one of two priorities. When high priority processes are active, no low priority processes are allowed any processor time.

Communication between processes is handled through *channels*. These are point to point, synchronized, and no buffers are used, i.e. communication is carried out only when both processes are ready. A channel between two processes situated on the same transputer is simply an address in memory. A channel between two processes situated on separate transputers is the link mentioned earlier.

Occam is a programming language which has been developed specifically for transputers. Occam is based on the CSP-model (communicating sequential processes), see [Hoare] for more information on CSP. Although many traditional languages, like the C programming language, have been converted to transputers, Occam is still the most natural language for transputers because it has facilities for communication and parallel process handling similar to the transputer itself. Occam is the language we have used. The OCCAM compiler generates very efficient code and optimizes to a high degree.

A.2 The MEiKO Transputer System

The MEiKO transputer system is a multi-user system with 48 transputers using a SUN workstation front-end. Several users can simultaneously share the 48 transputers, though a single transputer can only be used by one user. Of the 48 transputers, 12 are equipped with 2 Mbyte of external memory while the 36 others have 1 Mbyte of external memory. The MEiKO system

itself occupies from 3 to 5 of the transputers, so it is never possible to allocate more than 45 transputers at one time.

The transputers can be connected with each other in an almost arbitrary way through a programmable switch-board. The interconnection is of course limited by the fact that a transputer has four links only. This switch-board does not significantly reduce the communication speed of the links.

We have not used all of the available 45 transputers, because we experienced a number of curiosities. Figure A.2 is equivalent to figure 4.9 of section 4.2.5. Graphs a, b, and c seem almost as we would expect. Graphs d and e, however, show that something is wrong when 42 or more transputers are used. The error seems to get worse with an increased amount of computation. Hence, in figure A.2f we have set the number of units per processor in each of the layers to 10, i.e. we make the calculation part of the simulation so large that the communication part is insignificant. This is done in order to test whether it is speed of calculation or the communication between the transputers which is the problem.

This should result in a totally linear graph. And so it is with the use of up to 41 processors, but then there is a significant jump in the graph. Our only conclusion to this is: Four of the transputers are slower than the rest, perhaps they are not T800-30 but T800-20. We have not been able to verify this hypothesis. Due to this problem, we only use up to 40 transputers in our simulations,

We experienced another curiosity. When we allocate one transputer and run a sequential program on this transputer we obtain a certain running time. When we allocate 2 or 3 transputers and run the same sequential program again on just one of the transputers we naturally get the same running time. However if we allocate more than 3 transputers and run the very same sequential program again on one of these transputers, the program now runs approximately 25% faster. Hence when we run sequential programs we always allocate 4 transputers.

The two curiosities may be connected, though we have not been able to determine such a connection. In all our runs we have simply avoided the curiosities. Quite late in the project we learned that one of the boards (the transputers are placed on boards containing 4 transputers each) has some sort of fault. This is probably the reason for our troubles.

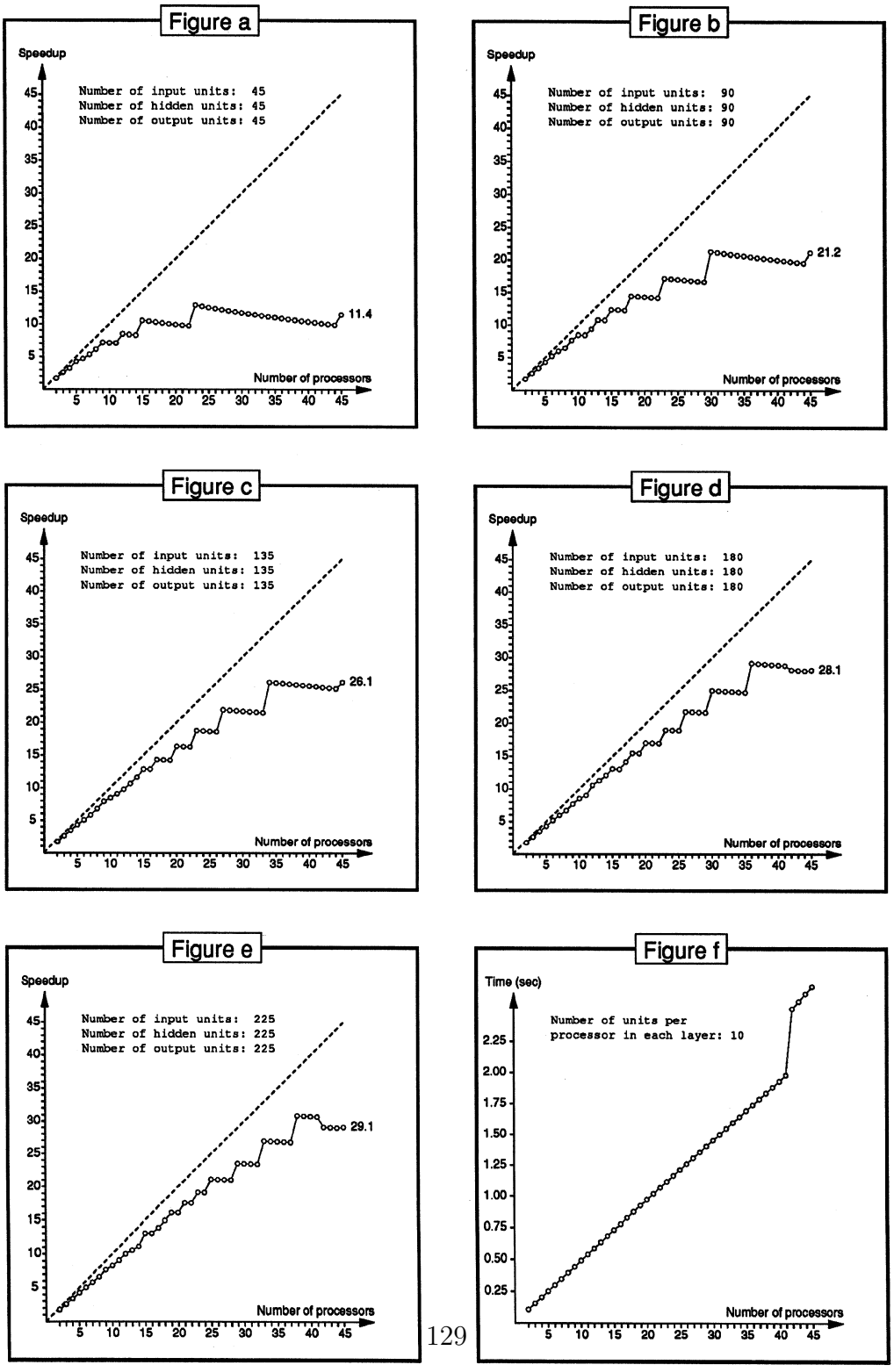

Figure A.2: Speedup graphs

A.3 Timings

Table A.1 is a summary of a table in an INMOS technical note [Inmos2] giving the times of the four basic floating point operations. The times are for double length arithmetic, i.e. `REAL64`.

Operation	Time for T800-30
Addition	233 nSec
Substraction	233 nSec
Multiplication	700 nSec
Division	1133 nSec

Table A.1: Speed of double length floating point operations

We also need to know the times for communicating arrays of `REAL64` on the external links. We have measured the communication times for arrays of varying sizes. The times are obtained in the following way: Both the sending and receiving transputers measure the communication times (and find the times identical). No calculation takes place concurrently with the communication. Furthermore, there is no software protocol for the communication. In this way we should be able to measure the fastest possible communication times.

The result is given in figure A.3. The figure shows that there is some constant initial contribution to each communication.

A.4 Concurrent Communication and Calculation

In this section we will briefly describe the benefits of performing communications and computations in parallel. For this purpose we have made a small parallel program, which is hopefully simple enough to allow us to understand what determines the results produced by the program.

Since any time used on communication will reduce the efficiency of the parallel algorithm, one can try to improve efficiency either by removing unnecessary communications, or by making each communication delay the main processor less. The latter can be achieved by inter-leaving the communications and calculations of the algorithm. This is possible, because of the

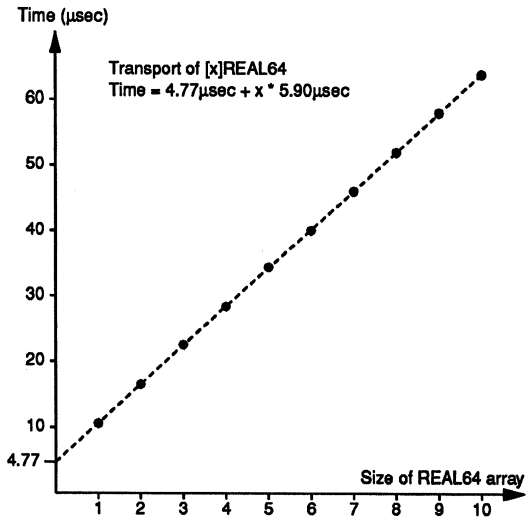

Figure A.3: Pure times for communicating `REAL64` arrays

on-chip link interface associated with each of the four links of the transputer. As mentioned in section A.1, each link interface is a small processor working independently of the main processor, and with its own direct access to the memory. Therefore, communications can be handled by the link interfaces while, at the same time, some of the computations of the algorithm are performed by the main processor, thereby reducing the cost of inter-processor communication.

Obviously, in order to be able to perform computations concurrently with communications, we will have to set up two parallel processes, one performing the computations, and the other handling the communications. But if we are to reduce running time, we must make sure that necessary computations are performed while the communications take place. One way to ensure this might be to give the communication process priority over the computation process, so that the link interface is initialized as quickly as possible. This way we expect the link interfaces to take over the handling of the communications after a short period of time, so that the main processor is left free to perform the computations.

In order to examine the effects of arranging the handling of communications and computations in different ways, we have made a number of runs with the program outlined in figure A.4. Shown here is the program running

on the root processor. The programs for the other processors are very similar to this program. The `token` communications are necessary for measuring the running time.

```
PROTOCOL Data.Protocol
  CASE
    Real; [COMM.SIZE]REAL64
    Token
:
Proc Administrator(CHAN OF ANY screen, CHAN OF Data.Protocol In, Out)
   ... Variables
  SEQ
     ... Initialize

    Timer ?  start.time

    Out !  Token
    In ?  CASE Token

    SEQ i = 0 FOR ITERATIONS
       ... Communication/computation task

    Out !  Token
    In ?  CASE Token

    Timer ?  stop.time
    write.full.string(screen, ``Time used: ''
    writwe.real64(screen, TicksToSecs(stop.time MINUS start.time), 0, 0)
    newline(screen)
:
```

Figure A.4: Program for testing different communication/computation arrangements

All results in this appendix were produced by running this program on 6 processors configured as a ring. In the tests, arrays consisting of 10 `REAL64` values are communicated (i.e. in the program `COMM.SIZE` is 10), and the computation consists of 50 `REAL64` multiplications and assignments (corresponding to a `CALC.SIZE` of 50 in the program).

In table A.2 we have given the running times for a number of different ways of handling communications and computations. The two first rows give the time required for performing either the communications or the computations. For each arrangement, we have given a reference to the figure showing

the associated piece of code. Note, that all times have been normalized after the running time of the fastest way of performing both communications and computations. As expected the fastest arrangement is to set up two parallel processes, one for communication and one for computation, and to give priority to the communication process.

Communication/Computation Task	Figure	Running Time
Pure communication	A.6	0.617
Pure computation	A.7	0.815
Communication priority	A.8	1.000
Sequential communication and computation	A.9	1.365
Computation priority	A.10	1.392
No priority	A.11	1.405

Table A.2: Running times for different communication/computation tasks

As shown in the table performing the communications and the computations sequentially is more than 36% slower than performing them in parallel with communications handled by a high-priority process and computations by a low-priority process. Note, that giving priority to the computation process actually produces a longer running time than performing communications and computations in sequence. The reason for this is probably, that because of the high-priority computation process, the communications are delayed until all computations have finished. So that, in reality, communications and computations are also performed sequentially in this case, but in addition to this we have the costs of creating extra parallel processes.

Obviously, performing communications and computations in parallel can at most save the time associated with the task requiring the smallest amount of time. However, by examining the table we can see that even though the 50 `REAL64` multiplications take longer time to perform than the communication does, the time required to perform the two tasks in parallel is more than 20% longer than the pure computation time.

It seems, therefore, that the cost of communicating cannot be removed altogether. In order to further investigate to what extent this is true, we made a number of experiments with varying communication and computation loads. Figure A.5 shows the results obtained. Figure *a* shows the results of a fixed amount of computation and a varying length of the arrays commu-

nicated. Figure *b* shows the results of keeping the length of the arrays fixed, and varying the amount of computation. In both figures the highlighted points represent runs with an array length of 10 `REAL64`s and a computation size of 50 `REAL64` multiplications (as in table A.2). Note, that in the figures *concurrent communication and computation* means that the two tasks were performed in parallel, and that the communication process was given priority over the computation process.

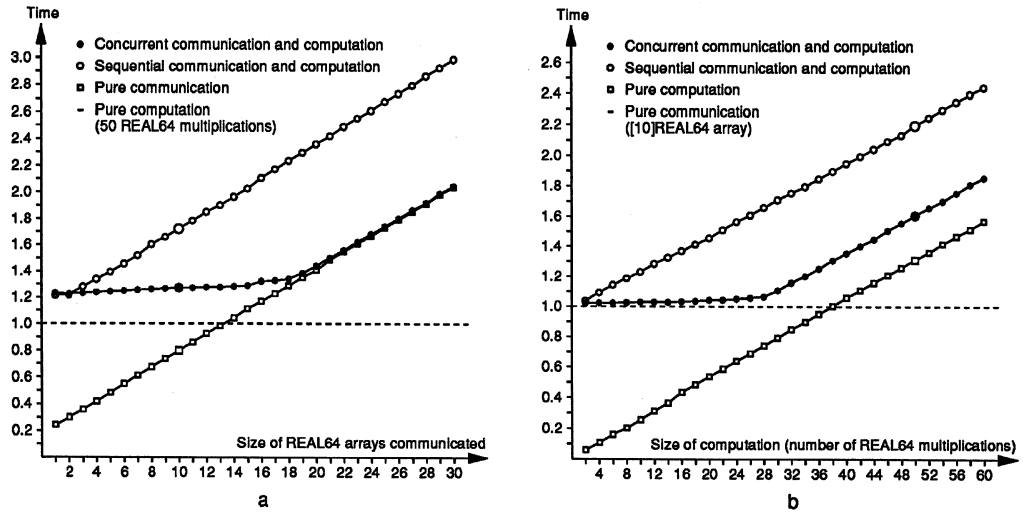

Figure A.5: Effects of concurrent communication and computation

The figures also show that when the pure computation time is smaller than the pure communication time the time required to perform the two tasks in parallel is determined by the communication time. In fact, the graphs show that in this case the computations do not slow down the communications at all.

Both figures show that even when the pure computation time is much longer than the pure communication time, performing the communications and the computations is parallel does *not* remove the cost of communicating entirely. However, if we look at figure *b*, we may observe that the difference between the time used by the concurrently performed communications and computations and the time used in pure computation constitutes a constant amount of absolute time. Therefore, the fraction of time used by communication becomes smaller and smaller when the amount of computation is increased, thus reducing the influence of communication on the efficiency of

the algorithm.

The main result of this appendix is that the cost of communication cannot be removed entirely, but it can be reduced significantly by performing the communications and the computations in parallel.

A.4.1 Communication/computation tasks

In this sub-section we show the different contents of the `Communication/computation task` folder of figure A.4 used in table A.2 and figure A.5. Both `comm.input` and `comm.output` are defined as `[COMM.SIZE] REAL64` arrays.

```
{{{  Communication/computation task
PAR
  In ?  CASE Real; comm.input
  Out !  Real; comm.output
}}}
```

Figure A.6: Only communications (no computation)

```
{{{  Communication/computation task
SEQ c = 0 FOR CALC.SIZE
  calc.var.1 := calc.var.2 * calc.var.3
}}}
```

Figure A.7: Only computation (no communications)

```
{{{  Communication/computation task
PRI PAR
  PAR
    In ?  CASE Real; comm.input
    Out !  Real; comm.output
  SEQ c = 0 FOR CALC.SIZE
    calc.var.1 := calc.var.2 * calc.var.3
}}}
```

Figure A.8: Parallel communications and computations (Communication priority)

```
{{{  Communication/computation task
SEQ
  PAR
    In ?  CASE Real; comm.input
    Out !  Real; comm.output
  SEQ c = 0 FOR CALC.SIZE
    calc.var.1 := calc.var.2 * calc.var.3
}}}
```

Figure A.9: Sequential communications and computations

```
{{{  Communication/computation task
PRI PAR
  SEQ c = 0 FOR CALC.SIZE
    calc.var.1 := calc.var.2 * calc.var.3
  PAR
    In ?  CASE Real; comm.input
    Out !  Real; comm.output
}}}
```

Figure A.10: Parallel communications and computations (computation priority)

```
{{{  Communication/computation task
PAR
    In ?  CASE Real; comm.input
    Out !  Real; comm.output
  SEQ c = 0 FOR CALC.SIZE
    calc.var.1 := calc.var.2 * calc.var.3
}}}
```

Figure A.11: Parallel communications and computations (no priority)

Appendix B

Program Listings

In this appendix the program listings are given. During the development of the programs we have used the *fold*-mechanism of the OPS-system in Odense. By means of this fold-mechanism is it possible to put a structure on the programs, such that it is easier to read them.

As can be seen in the programs on the following pages, a fold is represented as three dots followed by a character string. This character string is the name of the fold. Furthermore, it is the intention that the string should be an explanation of the contents of the fold. The fold can contain arbitrarily many other folds together with normal Occam code. An Occam program can then be regarded as a tree of folds, where each node is a named fold which can be succeeded by arbitrarily many sons. Each son is then a new fold.

We will present our programs by means of these folds. A fold is a framed box. The number in the top left corner is the number of the fold. The character string to the right of this number is the name of the fold. The number in the top right corner is the number of this fold's father in the fold-tree. When other folds exist inside a fold, the names of these folds will be succeeded by a number in brackets. The numbering of these folds is equivalent to a depth-first numbering of the fold-tree.

There are not many comments in the program listings. The reason for this is that a sensible structure of folds and the naming of these folds ought be explanation enough to understand a program.

B.1 Process Oriented Back-Propagation

1	Process Oriented Back-Propagation	0

```
...  Libraries (2)
...  Constants (3)
...  Variables (4)
...  PROC Environment(input.link, response.link, target.link) (5)
...  Simulator(input.link, response.link, target.link) (8)
...  Configure system (24)
```

2	Libraries	1

```
#Use linkaddr
#Use dblmath
#Use userio
#Use time
```

3	Constants	1

```
VAL learning.rate      IS  0.2(REAL64) :
VAL momentum           IS  0.9(REAL64) :
VAL low.weight         IS -0.3(REAL64) :
VAL high.weight        IS  0.3(REAL64) :
VAL BIAS.UNIT.ACTIVITY IS  0.1(REAL64) :

VAL INPUT.UNITS  IS 2:
VAL HIDDEN.UNITS IS 2:
VAL OUTPUT.UNITS IS 1:

VAL VAL NUMBER.OF.ITERATIONS IS 1000:
VAL VAL NUMBER.OF.PATTERNS IS     4:
```

4	Variables	1

```
[INPUT.UNITS]CHAN OF REAL64  input.link:
[OUTPUT.UNITS]CHAN OF REAL64 response.link:
[OUTPUT.UNITS]CHAN OF REAL64 target.link:
TIMER Timer:
INT Start, Stop:
```

5 PROC Environment(input.link,responce.link,target.link) 1

```
PROC Environment([CHAN OF REAL64 input.link,resonse.link,target.link)
  ... Variables    (6)
  SEQ
    ... Initialize    (7)
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
        pattern := i REM NUMBER.OF.PATTERNS
        PAR n = 0 FOR INPUT.UNITS
          input.link[n] !  input.pattern[pattern][n]
        PAR n = 0 FOR OUTPUT.UNITS
          response.link[n] ?  guess.activation[n]
        PAR n = 0 FOR OUTPUT.UNITS
          target.link[n] !  target.pattern[pattern][n]
:
```

6 Variables 5

```
[NUMBER.OF.PATTERNS][INPUT.UNITS]REAL64  input.pattern:
[NUMBER.OF.PATTERNS][OUTPUT.UNITS]REAL64 target.pattern:
[OUTPUT.UNITS]REAL64 guess.activity,  error:
INT pattern:
```

7 Initialize 5

```
input.pattern[0][0] := 0.0(REAL64)
input.pattern[0][1] := 0.0(REAL64)
input.pattern[1][0] := 0.0(REAL64)
input.pattern[1][1] := 1.0(REAL64)
input.pattern[2][0] := 1.0(REAL64)
input.pattern[2][1] := 0.0(REAL64)
input.pattern[3][0] := 1.0(REAL64)
input.pattern[3][1] := 1.0(REAL64)

target.pattern[0][0] := 0.1(REAL64)
target.pattern[1][0] := 0.9(RRAL64)
target.pattern[2][0] := 0.9(REAL64)
target.pattern[3][0] := 0.1(REAL64)
```

8	PROC Simulator(input.link, response.link, target.link)	1

```
PROC Simulator([]CHAN OF REAL64 input.link,response.link, target.link)
  ...   Variables  (9)
  ...   FUNCTION calculate.activity(net.input)   (10)
  ...   PROC input.neuron (number)  (11)
  ...   PROC hidden.neuron (number)  (12)
  ...   PROC output.neuron (number)  (18)

  PAR
    PAR i = 0 FOR INPUT.UNITS
      input.neuron(i)
    PAR i = 0 FOR HIDDEN.UNITS
      hidden.neuron(i)
    PAR i = 0 FOR OUTPUT.UNITS
      output.neuron(i)
:
```

9	Variables	8

```
[INPUT.UNITS][HIDDEN.UNITS]CHAN OF RRAL64 hidden.link:
[HIDDEN.UNITS][OUTPUT.UNITS]CHAN OF REAL64 output.link:
```

10	FUNCTION calculate.activity(net.input)	8

```
REAL64 FUNCTION caloulate.activity(VAL REAL64 net.input)
  REAL64 result:
  VALOF
    result := 1.0 (REAL64) / (1.0(REAL64) + DEXP(−net.input))
    RESULT result
:
```

11	PROC input.neuron(number)	8

```
PROC input.neuron(VAL INT number)
  REAL64 activity:
  SEQ i = 0 FOR NUMBER.OF.ITERATIONS
    SEQ
      input.link[number] ?  activity
      PAR j = 0 FOR HIDDEN.UNITS
        hidden.link[nmber][j] !  activity
:
```

12 PROC hidden.neuron(number) 8

```
PROC hidden.neuron(VAL INT number)
  ... Variables (13)
  SEQ
    ... Initialize (14)
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
        ... Propagate activity (15)
        ... Calculate weight changes (16)
        ... Change weights (17)
:
```

13 Variables 12

```
REAL64 activity, total.delta, delta, net.input, bias.weight, bias.change:
[INPUT.UNITS]REAL64 input.activity, weight, weight.change:
[OUTPUT.UNITS]REAL64 weighted.delta:
```

14 Initialize 12

```
INT64 seed:
REAL64 ran:
SEQ
  seed := (INT64 number)
  SEQ i = 0 FOR INPUT.UNITS
    SEQ
      ran, seed := DRAN(seed)
      weight[i] := low.weight + (ran * (high.weight - low.weight))
      weight.change[i] := 0.0(REAL64)
ran, seed := DRAN(seed)
bias.weight := low.weight + (ran * (high.weight - low.weight))
bias.change := 0.0(REAL64)
```

15	Propagate activity	12

```
PAR j = 0 FOR INPUT.UNITS
  hidden.link[j][number] ?  input.activity[j]
net.input := bias.weight * BIAS.UNIT.ACTIVITY
SEQ j = 0 FOR INPUT.UNITS
  net.input := net.input + (weight[j] * input.activity[j])
activity := calculate.activation(net.input)
PAR j = 0 FOR OUTPUT.UNITS
  output.link [number][j] !  activity
```

16	Calculate weight changes	12

```
PAR j = 0 FOR OUTPUT.UNITS
  output.link[number][j] ?  weighted.delta[j]
total.delta := 0.0(REAL64)
SEQ j = 0 FOR OUTPUT.UNITS
  total.delta := total.delta + weighted.delta[j]
delta := (total.delta * activity) * (1.0(REAL64) - activation)
SEQ j = 0 FOR INPUT.UNITS
  weight.change[j] := (momentum * weight.change[j]) +
                      (learning.rate * (delta * input.activity[j]))
bias.change := (momentum * bias.change) +
               (learning.rate * (delta * BIAS.UNIT.ACTIVITY))
```

17	Change weights	12

```
SEQ j = 0 FOR INPUT.UNITS
   weight[j] := weight[j] + weight.change[j]
bias.weight := bias.weight + bias.change
```

18	PROC output.neuron(number)	8

```
PROC output.neuron(VAL INT number)
   ... Variables (19)
  SEQ
     ... Initialize (20)
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
         ... Propagate activity (21)
         ... Calculate weight changes (22)
         ... Change weights (23)
:
```

19	Variables	18

```
REAL64 activity, net.input, bias.weight, bias.change, delta, target:
[INPUT.UNITS]REAL64 input.activity, weight, weight.ohange:
[HIDDEN.UNITS]REAL64 hidden.activity, hidden.error, weight, weight.change:
```

20	Initialize	18

```
INT64 seed:
REAL64 ran:
SEQ
  seed := (INT64 number)
  SEQ i = 0 FOR HIDDEN.UNITS
    SEQ
      ran, seed := DRAN(seed)
      weight[i] := low.weight + (ran * (high.weight - low.weight))
      weight.change[i] := 0.0(REAL64)
ran, seed := DRAN(seed)
bias.weight := low.weight + (ran * (high.weight - low.weight))
bias.change := 0.0(REAL64)
```

21	Initialize	18

```
PAR j = 0 FOR OUTPUT.UNITS
  output.link[number][j] ?  hidden.activity[j]
net.input := bias.weight * BIAS.UNIT.ACTIVITY
SEQ j = 0 FOR HIDDEN.UNITS
  net.input := net.input + (weight[j] * hidden.activity[j])
activity := calaulate.activation(net.input)
response.link[number] !  activation
```

22 Calculate weight changes 18

```
target.link [number] ?  target
delta := (target − activity) ∗
      (activity ∗ (1.0(REAL64) - activation))
PAR j = 0 FOR HIDDEN.UNITS
  SEQ
    hidden.error[j] := delta ∗ weight[j]
    output.link[j][number] !  hidden.error[j]
SEQ j = 0 FOR HIDDEN.UNITS
  weight.change[j] := (momentum ∗ weight.change[j]) +
                      (learning.rate ∗ (delta ∗ input.activity[j]))
bias.change := (momentum ∗ bias.change) +
               (learning.rate ∗ (delta ∗ BIAS.UNIT.ACTIVITY))
```

23 Change weights 18

```
SEQ j = 0 FOR HIDDEN.UNITS
  weight[j] := weight[j] + weight.change[j]
bias.weight := bias.weight + bias.change
```

24 Configure system 1

```
SEQ
  Timer ?  start

  PAR
    Environment(input.link, response.link, target.link)
    Simulator(input.link, response.link, target.link)

  Timer ?  stop
  write.full.string(screen, "Time used:  ")
  write.real64(screen, TicksToSecs(stop MINUS start), 0, 0)
  newline(screen)

  INT n:
  read.char(keyboard, n)
```

B.2 Sequential Back-Propagation - Pattern Updating

1	Standard Implementation of Back-Propagation	0

```
...  Libraries (2)
...  Constants (3)
...  Variables (4)
...  PROC Backprop()  (5)
...  Configure system  (26)
```

2	Libraries	1

```
#Use linkaddr
#Use dblmath
#Use userio
#Use time
```

3	Constants	1

```
VAL learning.rate      IS  0.2(REAL64) :
VAL momentum           IS  0.9(REAL64) :
VAL low.weight         IS -0.3(REAL64) :
VAL high.weight        IS  0.3(REAL64) :
VAL BIAS.UNIT.ACTIVITY IS  0.1(REAL64) :

VAL INPUT.UNITS  IS 2:
VAL HIDDEN.UNITS IS 2:
VAL OUTPUT.UNITS IS 1:

VAL VAL NUMBER.OF.ITERATIONS IS 1000:
VAL VAL NUMBER.OF.PATTERNS   IS    4:
```

4	Variables	1

```
TIMER timer
INT start, stop:
```

5 PROC Backprop () 1

```
PROC Backprop ()
   ... Variables  (6)
   ... FUNCTION calculate.activity (net.input)  (11)
  SEQ
     ... Initialize   (12)
     ... Create net   (13)
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
        pattern := i REM NUMBER.OF.PATTERNS
         ... Propagate activity   (16)
         ... Calculate weight change   (20)
         ... Change weight   (23)
:
```

6 Variables 5

```
...  Units (2)
...  Links (7)
...  Bias  (9)
...  Patterns  (10)

REAL64 net.input:
INT Pattern:
```

7 Units 6

```
[INPUT.UNITS]REAL64 input.unit.activity:
[HIDDEN.UNITS]REAL64 hidden.unit.activity:
[HIDDEN.UNITS]REAL64 hidden.unit.delta:
[OUTPUT.UNITS]REAL64 output.unit.activity:
[OUTPUT.UNITS]REAL64 output.unit.delta:
```

8 Links 6

```
[INPUT.UNITS][HIDDEN.UNITS]REAL64 hidden.link.weight:
[INPUT.UNITS][HIDDEN.UNITS]REAL64 hidden.link.change:
[HIDDEN.UNITS][OUTPUT.UNITS]REAL64 output.link.weight:
[HIDDEN.UNITS][OUTPUT.UNITS]REAL64 output.link.change:
```

9 Bias 6

```
[HIDDEN.UNITS]REAL64 hidden.bias.weight:
[HIDDEN.UNITS]REAL64 hidden.bias.change:
[OUTPUT.UNITS]REAL64 output.bias.weight:
[OUTPUT.UNITS]REAL64 output.bias.change:
```

10 Patterns 6

```
[NUMBER.OF.PATTERNS][INPUT.UNITS]REAL64   input.pattern :
[NUMBER.OF.PATTERNS][OUTPUT.UNITS]REAL64  target.pattern :
```

11 FUNCTION calculate.activity(net.input) 5

```
REAL64 FUNCTION calculate.activity(VAL REAL64 net.input)
  REAL64  result:
  VALOF
    result := 1.0 (REAL64)  /  (1.0(REAL64) + DEXP(-net.input))
    RESULT result
:
```

12 Initialize 5

```
input.pattern[0][0] := 0.0 (REAL64)
input.pattern[0][1] := 0.0 (REAL64)
input.pattern[1][0] := 0.0 (REAL64)
input.pattern[1][1] := 1.0 (REAL64)
input.pattern[2][0] := 1.0 (REAL64)
input,pattern[2][1] := 0.0 (REAL64)
input.pattern[3][0] := 1.0 (REAL64)
input.pattern[3][1] := 1.0 (REAL64)

target.pattern[0][0] := 0.1 (REAL64)
target.pattern[1][0] := 0.9 (REAL64)
target.pattern[2][0] := 0.9 (REAL64)
target.pattern[3][0] := 0.1 (REAL64)
```

13 Create net 5

```
INT64 seed:
REAL64 ran:
SEQ
  seed := 0 (INT64)
   ... Initialize weights between input and hidden layer  (14)
   ... Initialize weights between hidden and output layer (15)
```

14 Initialize weights between input and hidden layer 13

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    ran, seed := DRAN(seed)
    hidden.bias.weight[j] := low.weight +
                             (ran * (high.weight - low.weight))
    hidden.bias.change[j] := 0.0 (REAL64)
    SEQ i = 0 FOR INPUT.UNITS
      SEQ
        ran, seed := DRAN(seed)
        hidden.link.weight[i][j] := low.weight + (ran *
                                    (high.weight - low.weight))
        hidden.link.change[i][j] := 0.0 (REAL64)
```

15 Initialize weights between hidden and output layer 13

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    ran, seed := DRAN(seed)
    output.bias.weight[j] := low.weight +
                             (ran * (high.weight - low.weight))
    output.bias.change[j] := 0.0 (REAL64)
    SEQ i = 0 FOR HIDDEN.UNITS
      SEQ
        ran, seed := DRAN(seed)
        output.link.weight[i][j] := low.weight +
                                    (ran * (high.weight - low.weight))
        output.link.change[i][j] := 0.0 (REAL64)
```

16 Propagate activity 5

```
...  Set input activity  (17)
...  Calculate activity of hidden units  (18)
...  Calculate activity of output units  (19)
```

17 Set input activity 16

```
SEQ i = 0 FOR INPUT.UNITS
  input.unit.activity[i] := input.pattern[pattern][i]
```

18 Calculate activity of hidden units 16

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    net.input := hidden.bias.weight[j] * BIAS.UNIT.ACTIVITY
    SEQ i = 0 FOR HIDDEN.UNITS
      net.input := net.input +
                   (input.unit.activity[i] * hidden.link.weight[i][j])
      hidden.unit.activity[j] := calculate.activation(net.input)
```

19 Calculate activity of output units 16

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    net.input := output.bias.weight[j] * BIAS.UNIT.ACTIVITY
    SEQ i = 0 FOR HIDDEN.UNITS
      net.input := net.input +
                   (hidden.unit.activity[i] * output.link.weight[i][j])
    output.unit.activity[j] := calculate.activation(net.input)
```

20 Calculate weight changes 5

```
...  Calaulate weight changes between hidden and output units  (21)
...  Calculate weight changes between input and hidden units  (22)
```

21 Calculate weight changes between hidden and output units 20

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    output.unit.delta[j] :=
        target.pattern[pattern][j] − output.unit.activity[j]) ∗
        (output.unit.activity[j] ∗ (1.0(RRAL64) − output.unit.activation[j]))

    output.bias.change[j] := (momnentum ∗ output.bias.change[j]) +
          (learning.rate ∗ (output.unit.delta[j] ∗ BIAS.UNIT.ACTIVITY))
    SEQ i = 0 FOR HIDDEN.UNITS
      output.link.change[i][j] := (momentum ∗ output.link.change[i][j]) +
          (learning.rate ∗ (output.unit.delta[j] ∗ hidden.unit.activity[i]))
```

22 Calculate weight changes between input and hidden units 20

```
SEQ i = 0 FOR HIDDEN.UNITS
  SEQ
    hidden.unit.delta[i] := 0.0 (REAL64)
    SEQ j = 0 FOR OUTPUT.UNITS
      hidden.unit.delta[i] := hidden.unit.delta[i]) +
                              (output.unit.delta[i] ∗ output.link.weight[i][j])
    hidden.unit.delta[i] := (hidden.unit.delta[i] ∗
        hidden.unit.activity[i]) ∗ (1.0 (REAL64) - hidden.unit.activation[i])

    (hidden.bias.change[i] := (momentum ∗ hidden.bias.change[i]) +
        learning.rate ∗ (hidden.unit.delta[i] ∗ BIAS.UNIT.ACTIVITY))
    SEQ h = 0 FOR INPUT.UNITS
       hidden.link.change[h][i] := (momentum ∗ hidden.link.change[h][j])
                                   + (learning.rate ∗ (hidden.unit.delta[i]
                                   ∗ input.unit.activity[h]))
```

23 Propagate activity 5

```
...  Change weights between hidden and output units  (24)
...  Change weights between input and hidden units  (25)
```

24 Change weights between hidden and output units 23

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    SEQ i = 0 FOR HIDDEN.UNITS
      output.link.weight[i][j] := output.link.weight[i][j] +
                                  output.link.change[i][j])
    output.bias.weight[j] := output.bias.weight[j] +
                             output.bias.change[j]
```

25 Change weights between input and hidden units 23

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    SEQ i = 0 FOR INPUT.UNITS
      hidden.link.weight[i][j] := hidden.link.weight[i][j] +
                                  hidden.link.change[i][j])
    hidden.bias.weight[j] := hidden.bias.weight[j] + hidden.bias.change[j]
```

26 Configure system 1

```
SEQ
  Timer ?  start

  Backprop ()

  Timer ?  stop
  write.full.string(screen, "Time used:  ")
  write.real64(screen, TicksToSecs(stop MINUS start), 0, 0)
  newline (screen)

  INT n:
  read.char(keyboard, n)
```

B.3 Sequential Back-Propagation - Batch Version

1	Standard Implementation of Back-Propagation	0

```
...  Libraries (2)
...  Constants (3)
...  Variables (4)
...  PROC Backprop()  (5)
...  Configure system  (25)
```

2	Libraries	1

```
#Use linkaddr
#Use dblmath
#Use userio
#Use time
```

3	Constants	1

```
VAL learning.rate IS  0.2(REAL64) :
VAL momentum      IS  0.9(REAL64) :
VAL low.weight    IS -0.3(REAL64) :
VAL high.weight   IS  0.3(REAL64) :

VAL INPUT.UNITS IS 2:
VAL HIDDEN.UNITSIS 2:
VAL OUTPUT.UNITSIS 1:

VAL VAL NUMBER.OF.ITERATIONS IS 100:
VAL VAL NUMBER.OF.PATTERNS   IS   10:
VAL VAL NUMBER.OF.ITERATIONS IS 10 * BATCH.SIZE:
```

4	Variables	1

```
TIMER timer
INT start, stop, n
```

5 PROC Backprop () 1

```
PROC Backprop ()
   ... Variables   (6)
   ... FUNCTION calculate.activity (net.input)   (10)
  SEQ
     ...  COMMENT Load patterns
     ...  Create net   (11)
    SEQ i = 0 FOR NUMBER.OF.ITERATIONS
      SEQ
        pattern := i REM NUMBER.OF.PATTERNS
         ... Propagate activity   (14)
         ... Calculate weight change   (17)
        IF
           (i REM BATCH.SIZE) = 0
            ...  Update weights   (22)
           TRUE
             SKIP
:
```

6 Variables 5

```
...  Units  (7)
...  Links  (8)
...  Patterns  (9)

TIMER Timer:
INT start, stop, pattern:
```

7 Units 6

```
[HIDDEN.UNITS]REAL64 input.unit.activation:
[HIDDEN.UNITS]REAL64 hidden.unit.delta:
[HIDDEN.UNITS]REAL64 hidden.unit.error:

[OUTPUT.UNITS]REAL64 output.unit.activation:
[OUTPUT.UNITS]REAL64 output.unit.delta:
[OUTPUT.UNITS]REAL64 output.unit.error:
```

8 Links 6

```
[INPUT.UNITS+1][HIDDEN.UNITS]REAL64 hidden.link.weight:
[INPUT.UNITS+1][HIDDEN.UNITS]REAL64 hidden.link.change:
[INPUT.UNITS+1][HIDDEN.UNITS]REAL64 old.hidden.link.change:
[HIDDEN.UNITS+1][OUTPUT.UNITS]REAL64 output.link.weight:
[HIDDEN.UNITS+1][OUTPUT.UNITS]REAL64 output.link.change:
[HIDDEN.UNITS+1][OUTPUT.UNITS]REAL64 old.output.link.change:
```

9 Patterns 6

```
[NUMBER.OF.PATTERNS][INPUT.UNITS]REAL64 input.pattern:
[NUMBER.OF.PATTERNS][OUTPUT.UNITS]REAL64 target.pattern:
```

10 FUNCTION calculate.activity(net.input) 5

```
REAL64 FUNCTION calculate.activation(VAL REAL64 net.input)
  REAL 64  result:
  VALOF
    result := 1.0 (REAL64)  \  (1.0(REAL64) + DEXP(-net.input))
    RESULT result
:
```

11 Create net 5

```
INT64 seed:
REAL64 ran:
SEQ
  seed := 0 (INT64 1)
   ... Initialize weights between input and hidden layer  (12)
   ... Initialize weights between hidden and output layer (13)
```

12 Initialize weights between input and hidden layer 11

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      ran, seed := DRAN(seed)
      hidden.link.weight[i][j] :=
          low.weight + (ran * (high.weight - low.weight))
      hidden.link.change[i][j] := 0.0 (REAL64)
      old.hidden.link.change[i][j] := 0.0 (REAL64)
```

13 Initialize weights between hidden and output layer 11

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      ran, seed := DRAN(seed)
      output.bias.weight[j] := low.weight +
          low.weight + (ran * (high.weight - low.weight))
      output.link.change[j] := 0.0 (REAL64)
      old.output.link.change[i][j] := 0.0 (REAL64)
```

14 Propagate activity 5

```
REAL64 net:
SEQ
   ...  Calculate output from of hidden units (15)
   ...  Calculate output from of output units (16)
```

15 Calculate output from hidden units 14

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    net := hidden.link.weight[INPUT.UNITS][j]
    SEQ i = 0 FOR INPUT.UNITS
      net := net + (hidden.link.weight[i][j] * input.pattern[pattern][i])
    hidden.unit.activation[j] := calculate.activation(net)
```

16 Calculate output from output units 14

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    net := output.link.weight[HIDDEN.UNITS][j]
    SEQ i = 0 FOR HIDDEN.UNITS
      net := net + (hidden.unit.activation[i] * output.link.weight[i][j])
    output.unit.activation[j] := calculate.activation(net)
```

17 Calculate weight changes 5

```
SEQ
...  Calculate error on output units (18)
...  Calculate error on hidden units (19)
...  Calculate weight changes between hidden and output units (20)
...  Calculate weight changes between input and hidden units  (21)
```

18 **Calculate error on output units** **17**

```
SEQ i = 0 FOR OUTPUT.UNITS
  SEQ
    output.unit.error[i] := target.pattern[pattern][i] -
       output.uint.activation[i]
    output.unit.delta[i] :=(output.unit.error[i] *
                            output.unit.activation[i]) *
                            (1.0 (REAL64) - output.unit.activation[i])
```

19 **Calculate error on hidden units** **11**

```
SEQ i = 0 FOR HIDDEN.UNITS
  SEQ
    hidden.unit.error[i] := 0.0 (REAL64)
    SEQ j = 0 FOR OUTPUT.UNITS
      hidden.unit.error[i] := hidden.unit.error[i] +
                              (output.unit.delta[j] *
                               output.link.weight[i][j]))
    hidden.unit.delta[i] := (hidden.unit.error[i] *
                             hidden.unit.activation[i]) *
                             (1.0 (REAL64) - hidden.unit.activation[i])
```

20 **Calculate weight changes between hidden and output units** **17**

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    SEQ i = 0 FOR HIDDEN.UNITS
      output.link.change[i][j] := output.link.change[i][j] +
          (output.unit.delta[j] * hidden.unit.activation[i])
    output.link.change[HIDDEN.UNITS][j] :=
        output.link.change[HIDDEN.UNITS][j] + output.unit.delta[j]
```

21 **Calculate weight changes between input and hidden units** **17**

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    SEQ i = 0 FOR OUTPUT.UNITS
      hidden.link.change[i][j] := hidden.link.change[i][j] +
          (hidden.unit.delta[j] * output.pattern[pattern][i])
    hidden.link.change[INPUT.UNITS][j] :=
        hidden.link.change[INPUT.UNITS][j] + hidden.unit.delta[j]
```

22 Update weights 5

```
SEQ
    ... Change weights between hidden and output units  (23)
    ... Change weights between input and hidden units  (24)
```

23 Change weights between hidden and output units 22

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      output.link.change[i][j] := (momentum * old.output.link.change[i][j])
          + (learning.rate * output.link.change[i][j])
      output.link.weight[i][j] :=
          output.link.weight[i][j] + output.link.change[i][j]
old.output.link.change := output.link.change
SEQ j = = FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    output.link.change[i][j] := 0.0 (REAL64)
```

24 Change weights between input and hidden units 22

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      hidden.link.change[i][j] := (momentum * old.hidden.link.change[i][j])
          + (learning.rate * hidden.link.change[i][j])
      hidden.link.weight[i][j] :=
          hidden.link.weight[i][j] + hidden.link.change[i][j]
old.hidden.link.change := hidden.link.change
SEQ j = = FOR HIDDEN.UNITS
  SEQ i = 0 FOR OUTPUT.UNITS + 1
    hidden.link.change[i][j] := 0.0 (REAL64)
```

25	**Configur system**	**1**

```
SEQ
  Timer ?  start

  Backprop ()

  Timer ?  stop
  write.full.string(screen, "Time used:  ")
  write.real64(screen, TicksToSecs(stop MINUS start), 0, 0)
  newline(screen)

  read.char(keyboard, n)
```

B.4 Simple Data Partitioning Parallelization Using a Tree

The program given in the following is the program for the administrator (the program running on transputer 0). For this algorithm the program for the slaves is almost identical to that of the administrator, hence we do not give the entire program for the slaves. The only differences are within the folds *Initialize* (27) and *Simulate* (28). The modified folds for the slaves are given on page 170. Folds 27 and 28 should be replaced with folds 2 and 6 on page 170.

1 **Simple data partitioning parallelization using a tree - administrator** **0**

```
...  Libraries (2)
...  Constants (3)
...  Protocol  (4)
...  PROC Administrator(Left.out, Left.In, Right.Out, Right.In)  (5)
...  Configure transputer 0  (32)
```

2 **Libraries** **1**

```
#Use time
#Use linkaddr
#Use userio
#Use dblmath
#Use interf
```

3 Constants 1

```
VAL BATCH.SIZE IS 15:

VAL VAL NUMBER.OF.PROCESSORS IS 15

VAL VAL NUMBER.OF.BATH.ITER IS 10:
VAL VAL NUMBER.OF.PATTERNS  IS 15:

VAL learning.rate IS   0.1(REAL64) :
VAL momentum      IS   0.9(REAL64) :
VAL low.weight    IS - 0.5(REAL64) :
VAL high.weight   IS   0.5(REAL64) :

VAL INPUT.UNITS IS 50:
VAL HIDDEN.UNITS IS 50:
VAL OUTPUT.UNITS IS 50:

VAL PATTERNS.PR.PROCESSOR IS
    NUMBER.OF.PATTERNS +  NUMBER.OF.PROCESSORS - 1)) / NUMBER.OF.PROCESSORS:
VAL PATTERN.COUNT         IS PATTERNS.PR.PROCESSOR:
    BATCH.PR.PROCESSOR    IS
    (BATCH.SIZE + (NUMBER.OF.PROCESSORS - 1)) / NUMBER.OF.PROCESSORS:
VAL BATCH.COUNT           IS BATCH.PR.PROCESSOR:
```

4 Protocol 1

```
PROTOCOL Data.Protocol
    CASE
              Weights;   [INPUT.UNITS + 1][HIDDEN.UNITS]REAL64;
                         [HIDDEN.UNITS + 1][OUTPUT.UNITS]REAL64;
              Time; INT
:
```

5 PROC Administrator(Left.Out, Left.In, Right.Out, Right.In) 1

```
PROC Administrator(CHAN OF Data.Protocol Left.Out, Left.In, Right.Out)
                                         Right.In, Right.Out, Right.In)
   ... Variables  (6)
   ... Functions and procedures  (10)
  SEQ
     write.text.line(screen, "Conventional Back-Propagation")
     write.text.line(screen, "Back Parallel Version (TREE)")
     write.text.line(screen, "Last modification:  7/8 1991")
      ... Initialize    (26)
     - -  load.patterns ()
     Create.net ()
      ... Simulate    (27)
:
```

6 Variables 5

```
[4096]BYTE Garbage:
PLACE Garbage IN WORKSPACE:

 ...  Units    (7)
 ...  Links    (8)
 ...  Patterns (9)

INT batch.start, pattern, number.of.successors:
REAL64 net:

TIMER Timer:
INT start.time, stop.time:
```

7 Units 6

```
[HIDDEN.UNITS]REAL64 hidden.unit.activation:
[HIDDEN.UNITS]REAL64 hidden.unit.delta:
[HIDDEN.UNITS]REAL64 hidden.unit.error:

[OUTPUT.UNITS]REAL64 output.unit.activation:
[OUTPUT.UNITS]REAL64 output.unit.delta:
[OUTPUT.UNITS]REAL64 output.unit.error:
```

8	Links	6

```
[INPUT.UNITS + 1][HIDDEN.UNITS]REAL64  hidden.link.weight:
[INPUT.UNITS + 1][HIDDEN.UNITS]REAL64  hidden.link.change:
[INPUT.UNITS + 1][HIDDEN.UNITS]REAL64  old.hidden.link.change:
[INPUT.UNITS + 1][HIDDEN.UNITS]REAL64  hidden.link.left:
[INPUT.UNITS + 1][HIDDEN.UNITS]REAL64  hidden.link.right:
[HIDDEN.UNITS + 1][OUTPUT.UNITS]REAL64 output.link.weight:
[HIDDEN.UNITS + 1][OUTPUT.UNITS]REAL64 output.link.change:
[HIDDEN.UNITS + 1][OUTPUT.UNITS]REAL64 old.output.link.change:
[HIDDEN.UNITS + 1][OUTPUT.UNITS]REAL64 output.link.left:
[HIDDEN.UNITS + 1][OUTPUT.UNITS]REAL64 output.link.right:
```

9	Patterns	6

```
[NUMBER.OF.PATTERNS][INPUT.UNITS]REAL64  input.pattern:
[NUMBER.OF.PATTERNS][OUTPUT.UNITS]REAL64 target.pattern:
```

10	Functions and procedures	5

```
-- PROC load.patterns()
 ...   PROC create.net()   (11)
 ...   FUNCTION calculate.activation(net.input)   (14)
 ...   PROC propagate.activation(pattern)   (15)
 ...   PROC calculate.weight.changes(pattern)   (18)
 ...   PROC update.weights()  (23)
```

11	PROC create.net()	10

```
PROC create.net()
  INT64 seed:
  REAL64 ran;
  SEQ
     ...  Initialize weights between input and hidden layer   (12)
     ...  Initialize weights between hidden and output layer   (13)
:
```

12 Initialize weights between input and hidden layer 11

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    seed := (INT64 (j + 1) )
    SEQ i = 0 FOR INPUT.UNITS + 1
      SEQ
        ran, seed := DRAN(seed)
        hidden.link.weight[i][j] :=
            low.weight + (ran * (high.weight - low.weight))
        hidden.link.change[i][j] := 0.0(REAL64)
        old.hidden.link.change[i][j] := 0.0(REAL64)
```

13 Initialize weights between hidden and output layer 11

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    seed := (INT64 (j + 1) )
    SEQ i = 0 FOR INPUT.UNITS + 1
      SEQ
        ran, seed := DRAN(seed)
        output.link.weight[i][j] :=
            low.weight + (ran * (high.weight - low.weight))
        output.link.change[i][j] := 0.0(REAL64)
        old.output.link.change[i][j] := 0.0(REAL64)
```

14 FUNCTION calculate.activation(net.input) 10

```
REAL64 FUNCTION calculate.activation(VAL REAL64 net.input)
  REAL 64  result:
  VALOF
    result := 1.0 (REAL64)  /  (1.0(REAL64) + DEXP(-net.input))
    RESULT result
:
```

15 PROC propagate.activation(pattern) 10

```
PROC create.net()
PROC propagate.activation(VAL INT pattern)
  REAL64 net:
  SEQ
     ...  Calculate output from hidden units   (16)
     ...  Calculate output from output units   (17)
:
```

16 Calculate output from hidden units **15**

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    net := hidden.link.weight[INPUT.UNITS][j]
    SEQ i = 0 FOR INPUT.UNITS
      net := net + (hidden.link.weight[i][j] * input.pattern[pattern][i])
    hidden.unit.activation[j] := calculate.activation(net)
```

17 Calculate output from output units **15**

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    net := hidden.link.weight[HIDDEN.UNITS][j]
    SEQ i = 0 FOR HIDDEN.UNITS
      net := net + (hidden.unit.activation[i] * output.link.weight[i][j])
    output.unit.activation[j] := calculate.activation(net)
```

18 PROC calculate.weight.changes(pattern) **10**

```
PROC calculate.weight.changes(VAL INT pattern)
  SEQ
    ...  Calculate error on output units   (19)
    ...  Calculate error on hidden units   (20)
    ...  Calculate weight changes between hidden and output units   (21)
    ...  Calculate weight changes between input and hidden units   (22)
:
```

19 Calculate error on output units **18**

```
SEQ i = 0 FOR OUTPUT.UNITS
  SEQ
    output.unit.error[i] := target.pattern[pattern][i] -
                            output.unit.activation[i]
    output.unit.delta[i] := (output.unit.error[i] *
                            output.unit.activation[i]) *
                            (1.0 (REAL64) - output.unit.activation[i])
```

20 Calculate error on hidden units 18

```
SEQ i = 0 FOR HIDDEN.UNITS
  SEQ
    hidden.unit.error[i] := 0.0 (REAL64)
    SEQ j = 0 FOR OUTPUT.UNITS
      hidden.unit.error[i] := hidden.unit.error[i] + (output.unit.delta[j]
                               * output.link.weight[i][j]))
    hidden.unit.delta[i] := (hidden.unit.error[i] *
                             hidden.unit.activation[i]) *
                             (1.0 (REAL64) - hidden.unit.activation[i])
```

21 Calculate weight changes between hidden and output units 18

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    SEQ i = 0 FOR HIDDEN.UNITS
      output.link.change[i][j] := output.link.change[i][j] +
           (output.unit.delta[j] * hidden.unit.activation[i])
    output.link.change[HIDDEN.UNITS][j] :=
         output.link.change[HIDDEN.UNITS][j] + output.unit.delta[j]
```

22 Calculate weight changes between input and hidden units 18

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    SEQ i = 0 FOR INPUT.UNITS
      hidden.link.change[i][j] := hidden.link.change[i][j] +
           (hidden.unit.delta[j] * input.pattern[pattern][i])
    hidden.link.change[INPUT.UNITS][j] :=
      hidden.link.change[INPUT.UNITS][j] + hidden.unit.delta[j]
```

23 PROC update weights() 10

```
PROC update.weights()
  SEQ
    ...  Change weights between hidden and output units  (24)
    ...  Change weights between input and hidden units  (25)
```

24 **Change weights between hidden and output units** **23**

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      output.link.change[i][j] := (momentum * old.output.link.change[i][j]) +
          (learning.rate * output.link.change[i][j])
      output.link.weight[i][j] :=
          output.link.weight[i][j] + output.link.change[i][j]
old.output.link.change := output.link.change
SEQ j = = FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    output.link.change[i][j] := 0.0 (REAL64)
```

25 **Change weights between input and hidden units** **23**

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      hidden.link.change[i][j] := (momentum * old.hidden.link.change[i][j]) +
           (learning.rate * hidden.link.change[i][j])
      hidden.link.weight[i][j] :=
           hidden.link.weight[i][j] + hidden.link.change[i][j]
old.hidden.link.change := hidden.link.change
SEQ j = = FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    hidden.link.change[i][j] := 0.0 (REAL64)
```

26 **Initialize** **10**

```
IF
   NUMBER.OF.PROCESSORS > 2
      number.of.successors := 2
   NUMBER.OF.PROCESSORS = 2
      number.of.successors := 1
   TRUE
      number.of.successors := 0
```

27 Simulate 5

```
Timer ?  start.time

batch.start := 0
SEQ batch.iter = 1 FOR NUMBER.OF.BATCH.ITER
  SEQ
     ... Distribute new weights   (28)
    SEQ b = batch.start FOR BATCH.COUNT
      SEQ
        pattern := b REM PATTERN.COUNT
        propagate.activation(pattern)
        calculate.weight.changes(pattern)
    batch.start := (batch.start + BATCH.COUNT) REM PATTERN.COUNT
     ... Collect partial gradients and update weights   (29)
Timer ?  stop.time
write.full.string(screen, "Time used: ")
write.real64(screen, TicksToSecs(stop.time MINUS start.time), 0, 0)
newline(screen)
newline(screen)
```

28 Distribute new weights 27

```
IF
  number.of.successors = 2
    PAR
      Left.Out !  Weights; hidden.link.weight; output.link.weight
      Right.Out !  Weights; hidden.link.weight; output.link.weight
    number.of.successors = 1
      Left.Out !  Weights; hidden.link.weight; output.link.weight
  TRUE
    SKIP
```

29 Collect partial gradients and update weights 27

```
IF
  number.of.successors = 2
    SEQ
      PAR
        Left.In ?  CASE Weights; hidden.link.left; output.link.left
        Right.In ?  CASE Weights; hidden.link.right; output.link.right
       ... change := change + (left + right)   (30)
  number.of.successors = 1
    SEQ
      Left.In ?  CASE Weights; hidden.link.left; output.link.left
       ... change := change + left   (31)
  TRUE
    SKIP
update.weights()
```

30 change := change + (left + right) 29

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      output.link.change[i][j] := output.link.change[i][j] +
          output.link.left[i][j] + output.link.right[i][j]
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      hidden.link.change[i][j] := hidden.link.change[i][j] +
          (hidden.link.left[i][j] + hidden.link.right[i][j])
```

31 change := change + left 29

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      output.link.change[i][j] :=
          output.link.change[i][j] + output.link.left[i][j]
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      hidden.link.change[i][j] :=
          (hidden.link.change[i][j] + hidden.link.left[i][j])
```

32	**Configure transputer 0**	**1**

```
CHAN OF Data.Protocol Left.Out, Left.In, Right.Out, Right.In:
PLACE Left.Out AT Link1out:
PLACE Left.In AT Link1in:
PLACE Right.Out AT Link3out:
PLACE Right.In AT Link3in:
Addnistrator(Left.Out, Left.In, Right.Out, Right.In)
```

1	Simple data partitioning parallelization using a tree - slave	0

```
... Initialize   (2)
... Simulate   (6)
```

2	Initialize	1

```
... Calculate number of successors   (3)
... Calculate number of patterns for this prooessor   (4)
... Calculate batch size for this processor   (5)
```

3	Calculate number of successors	2

```
IF
  ((2 * node .number) + 1) <= NUMBER.OF.PROCRSSORS
    number.of.successors := 2
  (2 * node.number) <= NUMRER.OF.PROCESSORS
    number.of.successors := 1
  TRUE
    number.of.successors := 0
```

4	Calculate number of patterns for this processor	2

```
IF
  (NUMBER.OF.PATTERNS REM NUMBER.OF.PROCESSORS) = 0
    PATTERN.COUNT := PATTERNS.PR.PROCESSOR
  (NUMBER.OF.PATTERNS RRM NUMBER.OF.PROCESSORS) > (node.number - 1)
    PATTERN.COUNT := PATTERNS.PR.PROCESSOR
  TRUE
    PATTERN.COUNT := PATTERNS.PR.PROCESSOR - 1
```

5	Calculate batch size for this processor	2

```
IF
  (BATCH.SIZE REM NUMBER.OF.PROCESSORS) = 0
    PATTERN.COUNT := BATCH.PR.PROCESSOR
  (BATCH.SIZE RRM NUMBER.OF.PROCESSORS) > (node.number - 1)
    BATCH.COUNT := BATCH.PR.PROCESSOR
  TRUE
    BATCH.COUNT := BATCH.PR.PROCESSOR - 1
```

6 Simulate 1

```
batch.start := 0
 SEQ batch.iter = 1 FOR NUMBER.OF.BATCH.ITER
  SEQ
     ... Reset change   (7)
     ... Receive and distribute new weights   (8)
    SEQ b = batch.start FOR BATCH.COUNT
      SEQ
        pattern := b REM PATTERN.COUNT
        propagate.activation(pattern)
        calculate.weight.changes(pattern)
    batch.start := (batch.start + BATCH.COUNT) REM PATTERN.COUNT
     ... Collect partial gradients   (10)
```

7 Reset change 6

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    output.link.change[i][j] := 0.0 (Real64)
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    hidden.link.change[i][j] := 0.0 (REAL64)
```

8 Receive and distribute new weights 6

```
Parent.In ?  CASE
  Weights; hidden.link.weight; output.link.weight
    SEQ
       ... Distribute weights   (9)
```

9 Distribute weights 8

```
IF
  number.of.successors = 2
    PAR
      Left.Out !  Weights; hidden.link.weight; output.link.weight
      Right.Out !  Weights; hidden.link.weight; output.link.weight
  number.of.successors = 1
    Left.Out !  Weights; hidden.link.weight; output.link.weight
  TRUE
    SKIP
```

10 Collect partial gradients 6

```
IF
  number.of.successors = 2
    SEQ
      PAR
        Left.In ?  CASE Weights; hidden.link.left; output.link.left
        Right.In ?  CASE Weights; hidden.link.right; output.link.right
     ... change := change + (left + right)   (11)
  number.of.successors = 1
    SEQ
      Left.In ?  CASE Weights; hidden.link.left; output.link.left
     ... change := change + left   (12)
  TRUE
    SKIP
Parent.Out !  Weights; hidden.link.change; output.link.change
```

11 change := change + (left + right) 10

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      output.link.change[i][j] := output.link.change[i][j] +
          output.link.left[i][j] + output.link.right[i][j]
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      hidden.link.change[i][j] := hidden.link.change[i][j] +
          (hidden.link.left[i][j] + hidden.link.right[i][j])
```

12 change := change + left 10

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      output.link.change[i][j] :=
          output.link.change[i][j] + output.link.left[i][j])
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      hidden.link.change[i][j] :=
          (hidden.link.change[i][j] + hidden.link.left[i][j])
```

B.5 Simple Data Partitioning Parallelization Using a Ring

The program for the ring implementation is almost identical to that of the tree implementation (see B.4). The only difference is within the fold *Collect and sum partial gradients*. This fold is given on page 174.

1 **Simple data partitioning parallelization using a ring** **0**

```
... Collect and mm partial gradients   (2)
```

2 **Initialize weights between input and hidden layer** **1**

```
INT in, work:
SEQ
  work =:  0
  in =:  1
  PAR
    In ?  CASE Weights:  hidden.link[work]:  output.link[work]
    Out !  Weights; hidden.link.change; output.link.change
  SEQ p = 0 FOR NUMBER.OF.PROCESSORS - 2
    SEQ)
      PRI PAR
        PAR
          In ?  CASE Weights; hidden.link[in]; output.link[in]
          Out !  Weights; hidden.link[work]; output.link[work]
        SEQ
           ... change := change + work (3)
      work := 1 - work
      in := 1 - in
   ... change := change + work (3)
```

3	**change := change + work**	**2**

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      output.link.change[i][j] :=
          output.link.change[i][j] + output.link[work][i][j]
SEQ j = 0 FOR HIDDEN.UNITS + 1
  SEQ i = 0 FOR INPUT.UNITS + 1
    SEQ
      hidden.link.change[i][j] :=
          hidden.link.change[i][j] + hidden.link[work][i][j]
```

B.6 An Advanced Batch Updating Implementation — Administrator

1 **An Advanced Implementation of the Data Partitioning Strategy - Administrator** **0**

2 **Libraries** **1**

```
#Use time
#Use linkaddr
#Use userio
#Use dblmath
```

3 **Constants** **1**

```
VAL learning.rate IS   0.1  (REAL64)
VAL momentum      IS   0.9  (REAL64)
VAL low.weight    IS - 0.5  (REAL64)
VAL high.weight   IS   0.5  (REAL64)

VAL VAL INPUT.UNITS  IS 50:
VAL VAL HIDDEN.UNITS IS 50:
VAL VAL OUTPUT.UNITS IS 50:

VAL NUMBER.OF.PROCESSORS IS  40:
VAL BATCH.SIZE           IS 800:
VAL NUMBER.OF.PATTERNS   IS 800:
VAL NUMBER.OF.BATCH.ITER IS  50:
VAL NUMBER.OF.SLAVES     IS NUMBER.OF.PROCESSORS - 1:
VAL BATCH.DELTA          IS  50:
VAL MAX.BATCH.COUNT      IS BATCH.SIZE / NUMBER.OF.PROCESSORS:
VAL MAX.PATTERN.COUNT    IS NUMBER.OF.PATTERNS / NUMBER.OF.PROCESSORS:
```

4 Protocol 1

```
PROTOCOL Data.Protocol
  CASE
    Weight; REAL64;
    Change; REAL64;
    Weight.n.Change;  REAL64;  REAL64
:
```

5 PROC Administrator(In, Out) 1

```
PROC Administrator(CHAN OF Data.Protocol In, Out))
   ... Variables  (6)
   ... Functions and procedures  (10)
  SEQ
     ... Initialize  (15)
    write.text.line(screen, "Conventional Back-Propagation")
    write.text.line(screen, "Pomerleau Parallel Version")
    write.text.line(screen, "Last modification:  25/7 1991")
    -- load.patterns ()
    create.net ()
     ... Simulate    (16)
:
```

6 Variables 5

```
[4096]BYTE Garbage:
PLACE Garbage IN WORKSPACE:

 ...  Units   (7)
 ...  Links   (8)
 ...  Patterns  (9)

REAL64 dummy, change, weight, zero.value:
REAL64 weight.output, change.work, change.output:
INT batch.start, pattern:

TIMER Timer:
INT start.time, stop.time:
INT MAX.BATCH.FOR.ADMIN, BATCH.PR.SLAVE, BATCH.FOR.ADMIN, BATCH.COUNT:
INT PATTERNS.FOR.ADMIN, PATTERN.COUNT
```

7 Units 6

```
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.net:
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.activity:
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.error:
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.delta:

[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.net:
[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.activity:
[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.error:
[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.delta:
```

8 Links 6

```
[INPUT.UNITS+1][HIDDEN.UNITS]REAL64 hidden.link.weight:
[INPUT.UNITS+1][HIDDEN.UNITS]REAL64 old.hidden.link.change:
[HIDDEN.UNITS+1][OUTPUT.UNITS]REAL64 output.link.weight:
[HIDDEN.UNITS+1][OUTPUT.UNITS]REAL64 old.output.link.change:
```

9 Patterns 6

```
[MAX.PATTERNS.COUNT][INPUT.UNITS]REAL64 input.pattern:
[MAX.PATTERNS.COUNT][OUTPUT.UNITS]REAL64 target.pattern:
```

10 Functions and procedures 5

```
-- PROC load.patterns()
 ...  PROC create.net()  (11)
 ...  FUNCTION calculate.activation(net.input)  (14)
```

11 PROC create.net() 10

```
PROC create.net()
  INT64 seed:
  REAL64 ran;
  SEQ
    seed := (INT64 1)
     ...  Initialize weights between input and hidden layer  (12)
     ...  Initialize weights between hidden and output layer  (13)
:
```

12	Initialize weights between input and hidden layer	11

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS
    SEQ
      ran, seed := DRAN(seed)
      hidden.link.weight[i][j] :=
          low.weight + (ran * (high.weight - low.weight))
      old.hidden.link.change[i][j] := 0.0(REAL64)
```

13	Initialize weights between hidden and output layer	11

```
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS + 1
    SEQ
      ran, seed := DRAN(seed)
      output.link.weight[i][j] :=
          low.weight + (ran * (high.weight - low.weight))
      old.output.link.change[i][j] := 0.0(REAL64)
```

14	FUNCTION calculate.activation(net.input)	10

```
REAL64 FUNCTION calculate.activation(VAL REAL64 net.input)
  REAL 64  result:
  VALOF
    result := 1.0 (REAL64)  /  (1.0(REAL64) + DEXP(-net.input))
    RESULT result
:
```

15 Initialize 5

```
MAX.BATCH.FOR.ADMIN := (100 * BATCH.SIZE) /
  ((BATCH.DELTA * NUMBER.OF.SLAVES) + 100)
BATCH.PR.SLAVE :=
  ((BATCH.SIZE - MAX.BATCH.FOR.ADMIN) i (NUMBER.OF.SLAVES - 1)) /
    NUMBER.OF.SLAVES
BATCH.FOR.ADMIN := BATCH.SIZE - (NUMBER.OF.SLAVES * BATCH.PR.SLAVE)

IF
  (BATCH.FOR.ADMIN < 0)
    SEQ
      BATCH.FOR.ADMIN := 0
      BATCH.PR.SLAVE :=
          (BATCH.SIZE + (NUMBER.OF.SLAVES - 1)) / NUMBER.OF.SLAVES
TRUE
  SKIP

PATTERNS.FOR.ADMIN := (NUMBER.OF.PATTERNS * BATCH.FOR.ADMIN) / BATCH.SIZE

BATCH.COUNT := BATCH.FOR.ADMIN
PATTERN.COUNT := PATTERNS.FOR.ADMIN
```

16 Simulate 5

```
zero.value := 0.0 (REAL64)
Timer ?  start.time

SEQ batch.iter = 1 FOR NUMBER.OF.BATCH.ITER
  PAR
     ... Sender   (17)
     ... Receiver   (30)

Timer ?  stop.time)
write.full.string(screen, "Time used:  "
write.real64(screen, TicksToSecs(stop.time MINUS start.time), 0, 0)
newline(screen)
```

17 Sender 16

```
SEQ
   ...  Forward pass   (18)
   ...  Backward pass   (23)
```

18 Forward pass 17

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    SEQ p = 0 FOR BATCH.COUNT
      hidden.unit.net[p][j] := 0.0(REAL64)
    SEQ i = 0 FOR INPUT.UNITS
       ... Calculate net input to hidden units  (19)
SEQ j = 0 FOR HIDDEN.UNITS
   ... Calculate bias input and activity of hidden units  (20)
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    SEQ p = 0 FOR BATCH.COUNT
      output.unit.net[p][j] := 0.0(REAL64)
    SEQ i = 0 FOR HIDDEN.UNITS
       ... Calculate net input to output units  (21)
SEQ j = 0 FOR OUTPUT.UNITS
   ... Calculate bias input and activity of output units  (22)
Out !  Weight; zero.value -- dummy
```

19 Calculate net input to hidden units 18

```
SEQ
  weight.output := hidden.link.weight[i][j]
  PRI PAR
    OUT ! Weight; weight.output
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
        pattern := (p + batch.start) REM PATTERN.COUNT
        hidden.unit.net[p][j] := hidden.unit.net[p][j] +
             (input.pattern[pattern][i] * hidden.link.weight[i][j])
```

20 Calculate bias input and activity of hidden units 18

```
SEQ
  weight.output := hidden.link.weight[INPUT.UNITS][j]
  PRI PAR
    OUT ! Weight; weight.output
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
        hidden.unit.net[p][j] :=
             hidden.unit.net[p][j] + hidden.link.weight[INPUT.UNITS][j]
        hidden.unit.activity[p][j] :=
             (calculate.activity(hidden.unit.net[p][j])
```

21 Calculate net input to output units 18

```
SEQ
  weight.output := output.link.weight[i][j]
  PRI PAR
    OUT ! Weight; weight.output
    SEQ p = 0 FOR BATCH.COUNT
      output.unit.net[p][j] := output.unit.net[p][j] +
           (hidden.unit.activity[p][i] * output.link.weight[i][j])
```

22 Calculate bias input and activity of output units 18

```
SEQ
  weight.output := output.link.weight[HIDDEN.UNITS][j]
  PRI PAR
    OUT ! Weight; weight.output
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
        output.unit.net[p][j] :=
             output.unit.net[p][j] + output.link.weight[HIDDEN.UNITS][j]
        (output.unit.activity[p][i] :=
             calculate.activity(output.unit.net[p][j])
```

23 Backward pass 17

```
...  Send weights between hidden and output units (24)
...  Send initial weight changes for hidden links (27)
```

24 Send weights between hidden and output units 23

```
change.work := 0.0 (REAL64) -- dummy value
SEQ j = 0 FOR OUTPUT.UNITS
   ...  Calculate error on output units and bias weight changes  (25)
Out !  Change; change.work

change.work := 0.0(REAL64) -- dummy value
SEQ i = 0 FOR HIDDEN.UNITS
  SEQ
    SEQ p = 0 FOR BATCH.COUNT
      hidden.unit.error[p][i] := 0.0(REAL64)
    SEQ i = 0 FOR OUTPUT.UNITS
       ... Calculate weight changes and hidden errors  (26)
Out !  Weight.n.Change; zero.value; change.work -- dummy weight
```

25 **Calculate error on output units and bias weight changes** **24**

```
SEQ
  change.output := change.work
  change.work := 0.0 (REAL64)
  PRI PAR
    OUT ! Change; change.output
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
        pattern := (p + batch.start) REM PATTERN.COUNT
        output.unit.error[p][j] :=
             (target.pattern[pattern][j] + output.unit.activity[p][j])
        output.unit.delta[p][j] :=
             (output.unit.error[p][j] * output.unit.activity[p][j]) *
             (1.0 (REAL64) - output.unit.activity[p][j])
        change.work := change.work + output.unit.delta[p][j]
```

26 **Calculate weight changes and hidden errors** **24**

```
SEQ
  weight.output := output.link.weight[i][j]
  change.output := change.work
  change.work := 0.0 (REAL64)
  PRI PAR
    Out !  Weight.n.Change; weight.output; change.output
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
        hidden.unit.error[p][i] := hidden.unit.error[p][i] +
             (output.unit.delta[p][j] * output.unit.weight[i][j])
             (1.0 (REAL64) - output.unit.activity[p][j])
        change.work := change.work +
             (output.unit.delta[p][j] * hidden.unit.activity[p][i])
```

27 **Send initial weight changes for hidden links** **23**

```
change.work := 0.0 (REAL)  -- dummy value
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
     ... Calculate deltas on hidden units  (28)
    SEQ i = 0 FOR INPUT.UNITS
       ... calculate weight changes  (29)
Out !  Change; change.work
```

28 Calculate deltas on hidden units 24

```
change.output := change.work
change.work := 0.0(REAL64)
  PRI PAR
    Out !  Change; change.output
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
           hidden.unit.delta[p][j] :=
                 (hidden.unit.error[p][i] * hidden.unit.activity[p][j]) *
                  (1.0 (REAL64) - hidden.unit.activity[p][j])
           change.work := change.work + hidden.unit.delta[p][j])
```

29 Calculate weight changes 24

```
SEQ
  change.output := change.work
  change.work := 0.0 (REAL64)
  PRI PAR
    Out !  Change; change.output
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
        pattern := (p + batch.start) REM PATTERN.COUNT
        change.work := change.work +
             (hidden.unit.delta[p][j] * input.pattern[pattern][i])
```

30 Receiver 16

```
SEQ
   ...  Forward pass  (31)
   ...  Backward pass  (32)
```

31 Forward pass 30

```
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS
    In ?  CASE Weight; weight
SEQ j = 0 FOR HIDDEN.UNITS
  In ?  CASE Weight; weight
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ i = 0 FOR HIDDEN.UNITS
    In ?  CASE Weight; weight
SEQ j = 0 FOR OUTPUT.UNITS
  In ?  CASE Weight; weight
In ?  CASE Weight; weight -- dummy
```

32 Backward pass 30

```
...  Calculate new weights between hidden and output units  (33)
...  Calculate new weights between input and hidden units   (34)
```

33 Calculate new weights between hidden and output units 32

```
In ?  CASE Change; change -- dummy
SEQ j = 0 FOR OUTPUT.UNITS
  SEQ
    In ?  CASE Change; change
    change := (momentum * old.output.link.change[HIDDEN.UNITS] [j]) +
      (learning.rate * change)
    output.link.weight[HIDDEN.UNITS][j] :=
        output.link.weight[HIDDEN.UNITS][j] + change
      old.output.link.change[HIDDEN.UNITS][j] := change
In ?  CASE Weight.n.Change; weight; change -- dummy
SEQ i = 0 FOR HIDDEN.UNITS
  SEQ j = 0 FOR OUTPUT.UNITS
    SEQ
      In ?  CASE Weight.n.Change; weight; change
      change := (momentum * old.output.link.change[i][j]) +
        (learning.rate * change)
      output.link.weight[i] [j] :=
        output.link.weight[i][j] + change
      old.output.link.change[i][j] := change
```

34	Calculate new weights between input and hidden units	32

```
In ?  CASE Change; change -- dummy
SEQ j = 0 FOR HIDDEN.UNITS
  SEQ
    In ?  CASE Change; change
    change := (momentum * old.hidden.link.change[INPUT.UNITS] [j]) +
        (learning.rate * change)
    hidden.link.weight[INPUT.UNITS][j] :=
        hidden.link.weight[INPUT.UNITS][j] + change
      old.hidden.link.change[INPUT.UNITS][j] := change
      SEQ i = 0 FOR INPUT.UNITS
        SEQ
          In ?  CASE Change; change
          change := (momentum * old.hidden.link.change[i][j]) +
                (learning.rate * change)
          hidden.link.weight[i] [j] :=
                hidden.link.weight[i][j] + change
        old.hidden.link.change[i][j] := change
```

35	Configure transputer 0	1

```
CHAN OF Data.Protocol In, Out:
PLACE In AT Link1in:
PLACE Out AT Link3out:
Administrator(In, Out)
```

B.7 An Advanced Batch Updating Implementation — Slaves

1	An Advanced Implementation of the Data Partitioning Strategy - Slaves	0

```
...  Libraries (2)
...  Constants (3)
...  Protocol (4)
...  PROC Slave(In, Out) (5)
```

2	Libraries	1

```
#USE dblmath
```

3	Constants	1

```
VAL learning.rate IS  0.1  (REAL64)
VAL momentum      IS  0.9  (REAL64
VAL low.weight    IS -0.5  (REAL64)
VAL high.weight   IS  0.5  (REAL64)

VAL VAL INPUT.UNITS   IS 50:
VAL VAL HIDDEN.UNITS  IS 50:
VAL VAL OUTPUT.UNITS  IS 50:

VAL NUMBER.OF.PROCESSORS IS  40:
VAL BATCH.SIZE           IS 800:
VAL NUMBER.OF.PATTERNS   IS 800:
VAL NUMBER.OF.BATCH.ITER IS  10:
VAL NUMBER.OF.SLAVES     IS NUMBER.OF.PROCESSORS - 1:
VAL BATCH.DELTA          IS  103:
VAL MAX.BATCH.COUNT      IS
    (BATCH.SIZE + (NUMBER.OF.SLAVES - 1)) / NUMBER.OF.SLAVES:
VAL MAX.PATTERN.COUNT    IS
    (NUMBER.OF.PATTERNS + (NUMBER.OF.SLAVES - 1)) / NUMBER.OF.SLAVES:
```

4	Protocol	1

```
PROTOCOL Data.Protocol
  CASE
    Weight; REAL64;
    Change; REAL64;
    Weight.n.Change;  REAL64;   REAL64
:
```

5	PROC Slave(In, Out)	1

```
PROC Slave(CHAN OF Data.Protocol In, Out, VAL INT processor.number)
   ...  Variables  (6)
   ...  Functions and procedures  (9)
  SEQ
     --  load.patterns ()
     ...   Initialize   (22)
     ...   Simulate   (25)
:
```

6	Variables	5

```
[4096]BYTE Garbage:
PLACE Garbage IN WORKSPACE:

 ...  Units  (7)
 ...  Patterns  (8)

REAL64 dummy:
INT batch.start:
INT MAX.BATCH.FOR.ADMIN, BATCH.PR.SLAVE, BATCH.FOR.ADMIN:
INT PATTERNS.FOR.ADMIN, PATTERN.FOR.SLAVES
INT BATCH.COUNT, PATTERN.COUNT:
```

7 Units 6

```
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.net:
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.activity:
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.error:
[MAX.BATCH.COUNT][HIDDEN.UNITS]REAL64 hidden.unit.delta:

[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.net:
[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.activity:
[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.error:
[MAX.BATCH.COUNT][OUTPUT.UNITS]REAL64 output.unit.delta:
```

8 Patterns 6

```
[MAX.PATTERN.COUNT][INPUT.UNITS]REAL64 input.pattern:
[MAX.PATTERN.COUNT][OUTPUT.UNITS]REAL64 target.pattern:
```

9 Functions and procedures 6

```
-- PROC load.patterns()
 ...  FUNCTION calculate.activity(net.input)  (10)
 ...  PROC propagate.activity(batch.start)  (11)
 ...  PROC calculate.weight.changes(batch.start)  (16)
```

10 FUNCTION calculate.activity(net.input) 9

```
REAL64 FUNCTION calculate.activity(VAL REAL64 net.input)
  REAL64 result:
  VALOF
    result := 1.0 (REAL64) / (1.0(REAL64) + DEXP(-net.input))
    RESULT result
:
```

11 PROC propagate.activity(batch.start) 9

```
PROC propagate.activity(VAL INT batch.start)
  REAL64 weight.input, weight.work, weight.output:
  INT pattern:
  SEQ
    In ?  CASE Weight; weight.input
    SEQ j = 0 FOR HIDDEN.UNITS
      SEQ
        SEQ p = 0 FOR BATCH.COUNT
          hidden.unit.net[Ip][j] := 0.0 (REAL64)
        SEQ i = 0 FOR INPUT.UNITS
          ...   Calculate net inputs to hidden units   (12)
    SEQ j = 0 FOR HIDDEN.UNITS
      SEQ
        ... Calculate bias input and activity of hidden units   (13)
    SEQ j = 0 FOR OUTPUT.UNITS
      SEQ
        SEQ p = 0 FOR BATCH.COUNT
          output.unit.net[p][j] := 0.0 (REAL64)
        SEQ i = 0 FOR HIDDEN.COUNT
          ... Calculate net inputs to output units   (14)
    SEQ j = 0 FOR OUTPUT.UNITS
      SEQ
        ... Calculate bias input and activity of output units   (15)
    out !  Weight; weight.input -- dummy
:
```

12 Calculate net inputs to hidden units 11

```
SEQ
  weight.work := weight.input
  weight.output := weight.input
  PRI PAR
    PAR
      Out !  Weight; weight.output
      In ?  CASE Weight; weight.input
    SEQ p = 0 FOR BATCH.COUNT
      SEQ
        pattern := (p + batch.start) REM PATTERN.COUNT
        hidden.unit.net[p][j] := hidden.unit.net[p][j] +
              input.pattern[pattern][i] * weight.work
```

13 Calculate bias inputs and activity of hidden units 11

```
weight.work := weight.input
weight.output := weight.input
PRI PAR
  PAR
    Out !  Weight; weight.output
    In ?  CASE Weight; weight.input
  SEQ p = 0 FOR BATCH.COUNT
    SEQ
      hidden.unit.net[p][j] := hidden.unit.net[p][j] + weight.work
      hidden.unit.activity[p][j] :=
            calculate.activity(hidden.unit.net[p][j])
```

14 Calculate net inputs and output units 11

```
SEQ
  weight.work := weight.input
  weight.output := weight.input
  PRI PAR
    PAR
      Out !  Weight; weight.output
      In ?  CASE Weight; weight.input
    SEQ p = 0 FOR BATCH.COUNT
      output.unit.net[p][j] := output.unit.net[p][j] +
            (hidden.unit.activity[p][j] * weight.work)
```

15 Calculate bias inputs and activity of output units 11

```
weight.work := weight.input
weight.output := weight.input
PRI PAR
  PAR
    Out !  Weight; weight.output
    In ?  CASE Weight; weight.input
  SEQ p = 0 FOR BATCH.COUNT
    SEQ
      output.unit.net[p][j] := output.unit.net[p][j] + weight.work
      output.unit.activity[p][j] :=
            calculate.activity(output.unit.net[p][j])
```

16 PROC calculate.weight.changes(batch.start) 9

```
PROC caluclate.weight.changes(VAL INT batch.start)
  ...  Variables    (17)
  SEQ
    In ?  CASE Change; change.input -- dummy
    change.work := 0.0 (REAL64) -- dummy value
    SEQ j = 0 FOR OUTPUT.UNITS
      ...  calculate error on output units and bias weight changes   (18)
    change.output := change.work + change.input
    PAR
      Out !  Change; change.output
      In ?  CASE Weight.n.Change; weight.input; change.input
                                                    -- change = dummy
    SEQ i = 0 FOR INPUT.UNITS
      SEQ
        SEQ p = 0 FOR BATCH.COUNT
          hidden.unit.error[p][i] := 0.0 (REAL64)
        SEQ j = 0 FOR OUTPUT.UNITS
          ...   Calculate weight changes and hidden errors   (19)
    weight.output := weight.input
    change.output := change.work + change.input
    PAR
      Out !  Weight.n.Change; weight.output; change.output
      In ?  CASE Change; change.input -- dummy
    change.work := 0.0 (REAL64) -- dummy value
    SEQ j = 0 FOR HIDDEN.UNITS
      SEQ
        ... Calculate deltas on hidden units   (20)
    SEQ i = 0 FOR INPUT.UNITS
      SEQ
        ... Calculate weight changes   (21)
    out !  Change; change.output
:
```

17 Variables 16

```
REAL64 weight.input, weight.work, weight.output:
REAL64 change.input, change.work, change.output:
INT pattern:
```

18 Calculate error on output units and bias weight changes 16

```
SEQ
  change.output := change.work + change.input
  PRI PAR
    PAR
      Out !  Change; change.output
      In ?  CASE Change; change.input
    SEQ
      change.work := 0.0 (REAL64)
      SEQ p = 0 FOR BATCH.COUNT
        SEQ
          pattern := (p + batch.start) REM PATTERN.COUNT
          output.unit.error[p][j] :=
                (target.pattern[pattern][j] - output.unit.activity[p][j])
          output.unit.delta[p][j] :=
                (output.unit.error[p][j] * output.unit.activity[p][j]) *
                (1.0 (REAL64) - output.unit.activity[p][j])
          change.work := change.work + output.unit.delta[p][j]
```

19 Calculate weight changes and hidden errors 16

```
SEQ
  weight.work := weight.input
  weight.output := weight.input
  change.output := change.work + change.input
  PRI PAR
    PAR
      Out !  Weight.n.Change; weight.output; change.output
      In ?  CASE Weight.n.Change; weight.input; change.input
    SEQ
      change.work := 0.0 (REAL64)
      SEQ p = 0 FOR BATCH.COUNT
          SEQ
            hidden.unit.error [p][i] := hidden.unit.error[p][i] +
                  (output.unit.delta[p][j] * weight.work)
            change.work := change.work +
                  (output.unit.delta[p][j] * hidden.unit.activity[p][i])
```

20 Calculate deltas on hidden units 16

```
change.output := change.work + change.input
PRI PAR
  PAR
    Out !  Change; change.output
    In ?  CASE Change; change.input
  SEQ
    change.work := 0.0 (REAL64)
    SEQ p = 0 FOR BATCH.COUNT
        SEQ
          hidden.unit.delta [p][j] :=
                (hidden.unit.error[p][j] * hidden.unit.activity[p][j]) *
                (1.0 (REAL64) - hidden.unit.activity[p][j])
          (change.work := change.work + hidden.unit.delta[p][j]
```

21 Calculate weight changes 16

```
SEQ
  change.output := change.work + change.input
  PRI PAR
    PAR
      Out !  Change; change.output
      In ?  CASE Change; change.input
    SEQ
      change.work := 0.0 (REAL64)
      SEQ p = 0 FOR BATCH.COUNT
          SEQ
            pattern := (p + batch.start) REM PATTERN.COUNT
            change.work := change.work +
                  (hidden.unit.delta[p][j] * input.pattern[pattern][i])
```

22 Initialize 5

```
...  Calculate batch size for this processor (23)
...  Calculate number of patterns for this processor  (24)
batch.start := 0
```

23 Calculate batch size for this processor 22

```
MAX.BATCH.FOR.ADMIN := (100 * BATCH.SIZE) /
  ((BATCH.DELTA * NUMBER.OF.SLAVES) + 100)
BATCH.PR.SLAVE :=
  ((BATCH.SIZE - MAX.BATCH.FOR.ADMIN) + (NUMBER.OF.SLAVES - 1)) /
    NUMBER.OF.SLAVES
BATCH.FOR.ADMIN := BATCH.SIZE - (NUMBER.OF.SLAVES * BATCH.PR.SLAVE)

IF
  (BATCH.FOR.ADMIN < 0)
    SEQ
      BATCH.FOR.ADMIN := 0
      BATCH.PR.SLAVE :=
        (BATCH.SIZE + (NUMBER.OF.SLAVES - 1)) / NUMBER.OF.SLAVES
  TRUE
    SKIP

IF
  ((BATCH.SIZE - BATCH.FOR.ADMIN) REM NUMBER.OF.SLAVES) = 0
     BATCH.COUNT := BATCH.PR.SLAVE
  ((BATCH.SIZE - BATCH.FOR.ADMIN) REM NUMBER.OF.SLAVES) < processor.number
     BATCH.COUNT := BATCH.PR.SLAVE - 1
  TRUE
     BATCH.COUNT := BATCH.PR.SLAVE
```

24 Calculate number of patterns for this processor 22

```
PATTERNS.FOR.ADMIN := (NUMBER.OF.PATTERNS * BATCH.FOR.ADMIN) / BATCH.SIZE
PATTERNS.FOR.SLAVES := NUMBER.OF.PATTERNS - PATTERNS.FOR.ADMIN

(PATTERN.COUNT :=
    (PATTERNS.FOR.SLAVES + (NUMBER.OF.SLAVES - 1)) / NUMBER.OF.SLAVES
IF
  (PATTERNS.FOR.SLAVES REM NUMBER.OF.SLAVES) = 0
    SKIP
  (PATTERNS.FOR.SLAVES REM NUMBER.OF.SLAVES) < processor.number
     PATTERN.COUNT := PATTERN.COUNT - 1
  TRUE
     SKIP
```

25	**Simulate**	**5**

```
SEQ batch.iter = 1 FOR NUMBER.OF.BATCH.ITER
  SEQ
    propagate.activity(batch.start)
    calculate.weight.changes(batch.start)
    batch.start := (batch.start + BATCH.COUNT) REM PATTERN.COUNT
```

B.8 Matrix Multiplication Algorithm — Administrator

The program given in the following is the program for the administrator (transputer 0). The administrator is very simple. It just sends along what it receives. The faster the better.

1 **Matrix Multiplication Algorithm - Administrator** **0**

```
...  Libraries (2)
...  Constants (3)
...  Protocol (4)
...  PROC Administrator (Out, In) (5)
...  Configure transputer 0 (14)
```

2 **Libraries** **1**

```
#USE time
#USE linkaddr
#USE userio
#USE dblmath
#USE interf
```

3 **Constants** **1**

```
VAL PATTERNS.PR.PROCESSOR IS 15:
VAL SIZE                  IS  4:
VAL INPUT.PR.PROCESSOR    IS 15:
VAL HIDDEN.PR.PROCESSOR   IS 15:
VAL OUTPUT.PR.PROCESSOR   IS 15:
VAL INPUT.UNITS           IS INPUT.PR.PROCESSOR * SIZE:
VAL VAL HIDDEN.UNITS      IS HIDDEN.PR.PROCESSOR * SIZE:
VAL OUTPUT.UNITS          IS OUTPUT.PR.PROCESSOR * SIZE:
VAL NUMBER.OF.PATTERNS    IS PATTERNS.PR.PROCESSOR * SIZE:
VAL NUMBER.OF.ITERATIONS  IS 10:
```

4 Protocol 1

```
PROTOCOL Protocol
  CASE
    Input.Activity; [INPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Hidden.Activity; [HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Hidden.Weight; [HIDDEN.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64
    Output.Weight; [OUTPUT.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64
    Hidden.Dalta; [HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Output.Delta; [OUTPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Time
:
```

5 PROC Administrator(Out, In) 1

```
PROC Administrator(CHAN Op Protocol Out, In)
   ... Variables  (6)
  SEQ
     ... Initialize torus  (7)

    write.full.string (screen, "Number of prooessors:  ")
    write.int(screen, SIZE * SIZE, 0)
    newline(screen)

    Timer ?  start
    ...   Simulate  (8)
    In ?  CASE Time
    Timer ?  stop

    write.full.string(screen, "Time used:  ")
    write.real64(screen, TicksToSecs(stop MINUS start), 0, 0)
    INT n:
    read.char(keyboard, n)
:
```

6 Variables 5

```
[HIDDEN.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64 hidden.weight:
[OUTPUT.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64 output.weight:
[INPUT.PR.PROCESSOR] [PATTERNS.PR.PROCESSOR]REAL64 input.activity0:
[INPUT.PR.PROCRSSOR][PATTERNS.PR.PROCESSOR]REAL64 input.activity1:
[HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 hidden.activity0:
[HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 hidden.activity1:
[HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 net.input:
[OUTPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 output.delta:

TIMER Timer
INT start, stop:
```

7 Initialize torus 5

```
PAR
  In ?  CASE Input.Activity; input.activity0
  Out !  Input.Activity; input.activity1
```

8 Simulate 5

```
[SEQ iteration = 0 FOR NUMBER.OF.ITERATIONS
  SEQ
    ...   Calculate hidden unit activity  (9)
    ...   Calaulate output unit activity  (10)
    ...   Calculate hidden unit delta values  (11)
    ...   Calculate output weight changes  (12)
    ...   Calculate hidden weight changes  (13)
```

9 Calculate hidden unit activity 8

```
SEQ 1 = 0 FOR SIZE - 1
  SEQ
    In ?  CASE Input.Activity; input.activity0
    Out !  Input.Activity; input.activity0
```

10 Calculate output unit activity 8

```
SEQ i = 0 FOR SIZE - 1
  SEQ
    In ?  CASE Hidden.Activity; hidden.activity0
    Out !  Hidden.Activity; hidden.activity0
```

11	Calculate hidden unit delta values	8

```
SEQ d = 0 FOR SIZE - 1
  SEQ
    In ?  CASE Output.Weight:  output.weight
    Out !  Output.Weight; output.weight
    In ?  CASE Output.Delta; output.delta
    Out !  Output.Delta:  output.delta
```

12	Calculate net inputs to hidden units	11

```
In ?  CASE Hidden.Activity; hidden.activity0
SEQ i = 0 FOR SIZE - 1
  IF
    (i REM 2) = 0
      PAR
        In ?  CASE Hidden.Activity; hidden.activity1
        Out !  Hidden.Activity; hidden.activity0
    TRUE
      PAR
        In ?  CASE Hidden.Activity; hidden.activity0
        Out !  Hidden.Activity; hidden.activity1
IF
  (SIZEREM 2) = 0
    Out !  Hidden.Activity; hidden.activity1
  TRUE
    Out !  Hidden.Activity; hidden.activity0
```

13	Calculate hidden weights changes	8

```
In ?  CASE Input.Activity; input.aativity0
SEQ i = 0 FOR SIZE - 1
  IF
    (i REM 2) = 0
      PAR
        In ?  CASE Input.Aativity; input.activity1
        Out !  Input.Aativity; input.activity0
    TRUE
      PAR
        In ?  CASE Input.Aativity; input.activity0
        Out !  Input.Aativity; input.activity1
IF
  (SIZE REM 2) = 0
    Out !  Input.Aativity; input.activity1
  TRUE
    Out !  Input.Aativity; input.activity0
```

14	Configure transputer 0	1

```
CHAN OF Protocol In, Qut:
PLACE In AT Link1in:
PLACE Out AT Link3out:
Administrator(Out, In)
```

B.9 Matrix Multiplication Algorithm — Slaves

1	Matrix Multiplication Algorithm - Slave	0

```
...  Libraries (2)
...  Constants (3)
...  Protocol  (4)
...  PROC Slave(Up, Down, Right, Left, processor.number)  (5)
```

2	Libraries	1

```
#USE dblmath
```

3	Constants	1

```
VAL learning.rate  IS  0.3(REAL64):
VAL momentum       IS  0.9(REAL64):
VAL low.weight     IS -0.3(REAL64):
VAL heigh.weight   IS  0.3(REAL64):

VAL PATTERNS.PR.PROCESSOR IS 15:
VAL SIZE                  IS  4:
VAL INPUT.PR.PROCESSOR    IS 15:
VAL HIDDEN.PR.PROCESSOR   IS 15:
VAL OUTPUT.PR.PROCESSOR   IS 15:
VAL INPUT.UNITS           IS INPUT.PR.PROCESSOR * SIZE:
VAL VAL HIDDEN.UNITS      IS HIDDEN.PR.PROCESSOR * SIZE:
VAL OUTPUT.UNITS          IS OUTPUT.PR.PROCESSOR * SIZE:
VAL NUMBER.OF.PATTERNS    IS PATTERNS.PR.PROCESSOR * SIZE:
VAL NUMBER.OF.ITERATIONS  IS 10:
```

4	Protocol	1

```
PROTOCOL Protocol
  CASE
    Input.Activity; [INPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Hidden.Activity; [HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Hidden.Weight; [HIDDEN.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64
    Output.Weight; [OUTPUT.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64
    Hidden.Dalta; [HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Output.Delta; [OUTPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64
    Time
:
```

5	Slave(Up, Down, Right, Left, processor.number)	1

```
PROC Slave(CHAN OF Protocol Up, Down, Right, Left,
                                      VAL INT prooessor.number)
  ...  Variables (6)
  ...  Procedures and functions  (7)
  SEQ
     ... Initialize  (12)
    -- load.patterns ()
    create.net ()
    ... Initialize torus  (13)

    ...   Simulate  (14)

    ...   Terminate  (40)
:
```

6 Variables 5

```
[4096]BYTE garbage:
PLACE garbage IN WORKSPACE:

[SIZE][HIDDEN.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64 hidden.weight:
[HIDDEN.PR.PROCESSOR][INPUT.PR.PROCESSOR]REAL64 hidden.change:
[HIDDEN.PR.PROCRSSOR][INPUT.PR.PROCESSOR]REAL64 old.hidden.change:
[HIDDEN.PR.PROCESSOR]REAL64 hidden.bias:
[HIDDEN.PR.PROCESSOR]REAL64 hidden.bias.change:
[HIDDEN.PR.PROCESSOR]REAL64 old.hidden.bias.change:

[SIZE][OUTPUT.PR.PROCESSOR][HIDDEN.PR.PROCESSOR]REAL64 output.weight:
[OUTPUT.PR.PROCESSOR][HIDDEN.PR.PROCESSOR]REAL64 output.change:
[OUTPUT.PR.PROCESSOR][HIDDEN.PR.PROCESSOR]REAL64 old.output.change:
[OUTPUT.PR.PROCESSOR]REAL64 output.bias:
[OUTPUT.PR.PROCESSOR]REAL64 output.bias.change:
[OUTPUT.PR.PROCESSOR]RRAL64 old.output.bias.change

[SIZE][INPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 input.activity:
[SIZE][HIDDEN.PR.PROCESSOR] [PATTERNS.PR.PROCESSOR]REAL64 hidden.activity:
[SIZE][OUTPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 output.activity:

[HIDDEN.PR.PROCESSOR][PATTRENS.PR.PROCESSOR]REAL64 hidden.net:
[OUTPUT.PR.PROCPSSOR][PATTERNS.PR.PRCCESSOR]REAL64 output.net:

[OUTPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 target.pattern:

[SIZE][HIDDEN.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 hidden.delta:
[SIZE][OUTPUT.PR.PROCESSOR][PATTERNS.PR.PROCESSOR]REAL64 output.delta:

INT x, y
```

7 Procedures and functions 5

```
-- PROC load.patterns()
...   FUNCTION calculate.activity(net.input)  (8)
...   PROC create.net()  (9)
```

8 FUNCTION calculate.activity(net.input) 7

```
REAL64 FUNCTION calculate.activity(VAL REAL64 net.input)
    REAL64 result:
    VALOF
      result := 1.0(REAL64) / (1.0(REAL64) + DEXP(-net.input))
      RESULT result
:
```

9 PROC create.net() 7

```
PROC create.net()
  REAL64 ran:
  INT64 seed:
  SEQ d = 0 FOR SIZE
    SEQ
      ...   Initialize weights between input and hidden layer  (10)
      ...   Initialize weights between hidden and output layer  (11)
:
```

10 Initialize weights between input and hidden layer 9

```
SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
  SEQ seed := (INT64 ((processor.number + i) + 1))
    ran, seed := DRAN(seed)
    hidden.bias[i] := ran
    hidden.bias.change[i] := 0.0(REAL64)
    SEQ j = 0 FOR INPUT.PR.PROCESSOR
      SEQ
        ran, seed := DRAN(seed)
        hidden.weight[d][i][j] :=
              low.weight + (ran * (high.weight - low.weight))
        hidden.change[i][j] := 0.0(REAL64)
```

11	Initialize weights between hidden and out layer	9

```
SEQ i = 0 FOR OUTPUT.RR.PROCESSOR
  SEQ
    seed := (INT64 ((processor.number + i) + 1))
    ran, seed := DRAN(seed)
    output.bias[i] := ran
    output.bias.changa[i] := 0.0(REAL64)
    SEQ j = 0 FOR INPUT.PR.PROCESSOR
      SEQ
        ran, seed := DRAN(seed)
        output.weight[d][i][j] :=
              low.weight + (ran * (high.weight - low.weight))
        output.change[i][j] := 0.0(REAL64)
```

12	Initialize	5

```
x := (processor.nunber - 1) REM SIZE
y := (processor.number - 1) / SIZE
```

13	Initialize torus	5

```
PAR
  Up !  Input.Activity; input.activity[0]
  Down ?  CASE Input.Activity; input.activity[1]

PAR
  Right !  Hidden.Weight; hidden.weight[0]
  Left ?  CASE Hidden.Weight; hidden.weight[1]
```

14	Simulate	5

```
SEQ iteration = 0 FOR NUMBER.OF.ITERATIONS
  SEQ
    ... Propagate input unit activity  (15)
    ... Propagate hidden unit activity  (18)
    ... Calculate output unit delta values  (21)
    ... Calculate hidden unit delta values  (22)
    ... Calculate output weight changes  (30)
    ... Calculate hidden weight changes  (35)
```

15 Propagate input unit activity 14

```
... Distribute hidden weights and input unit activities  (16)
... Calculate hidden unit activities  (17)
```

16 Distribute hidden weights and input unit activities 15

```
SEQ i = 0 FOR SIZE - 1
  PAR
    PAR
      Right !  Hidden.Weight; hidden.weight[((x + SIZE) - i) REM SIZE]
      Left ?  CASEHidden.Weight;
                  hidden.weight[(((x + SIZE) - i) - 1) REM SIZE]
  PAR
      Up !  Input.Activity; input.activity[(y t i) REM SIZE]
      Down ?  CASE Input.Activity;
                  input.activity[((y + i) + 1) REM SIZE]
```

17 Calculate hidden unit activities 15

```
SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
  SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
    hidden.net[i][j] := hidden.bias[i]

SEQ d = 0 FOR SIZE
  SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
    SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
      SEQ k = 0 FOR INPUT.PR.PROCESSOR
        hidden.net[i][j] := hidden.net[i][j] +
          (hidden.weight[d][i][k] * input.activity[d][k][j])

SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
  SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
    hidden.activity[y][i][j] := calculate.activity(hidden.net[i][j])
```

18 Propagate hidden unit activity 14

```
... Distribute output weights and hidden unit activities  (19)
... Calculate output unit net inputs  (20)
```

19 Distribute output weights and hidden unit activities 15

```
SEQ i = 0 FOR SIZE - 1
  PAR
    PAR
      Right !  Output.Weight; output.weight[((x + SIZE) - i) REM SIZE]
      Left ?  CASE Output.Weight;
                   output.weight[(((x + SIZE) - i) - 1) REM SIZE]
  PAR
      Up !  Hidden.Activity; hidden.activity[(y t i) REM SIZE]
      Down ?  CASE Hidden.Activity;
                   hidden.activity[((y + i) + 1) REM SIZE]
```

20 Calculate output unit net inputs 18

```
SEQ i = 0 FOR OUTPUT.PR.PROCESSOR
  SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
    output.net[i][j] := output.bias[i]

SEQ d = 0 FOR SIZE
  SEQ i = 0 FOR OUTPUT.RR.PROCESSOR
    SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
      SEQ k = 0 FOR HIDDEN.PR.PROCESSOR
        output.net[i][j] := output.net[i][j] +
             (output.weight[d][i][k] * hidden.activity[d][k][j])

SEQ i = 0 FOR OUTPUT.PR.PROCESSOR
  SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
    output.activity[y][i][j] := calculate.activity(output.net[i][j])
```

21 Calculate output unit delta values 14

```
SEQ i = 0 FOR OUTPUT.PR.PROCESSOR
  SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
    output.delta[y][i][j] := (target.pattern[i][j]
       - output.activity[y][i][j]) * (output.activity[y][i][j] *
        (1.0(REAL64) - output.activity[y][i][j]))
```

22 Calculate hidden unit delta values 14

```
...  Initialize  (23)
...  Distribute weights  (24)
SEQ d = 0 FOR SIZE - 1
  PAR
    ...  Calaulate hidden unit error values  (25)
    SEQ
      ...  Send up output weights  (26)
      PAR
        ...  Distribute weights  (27)
        ...  Send up output delta values  (28)
...  Calculate hidden unit error values  (23)
SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
  SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
    hidden.delta[y][i][j] := hidden.delta[y][i][j] *
         (hidden.activity[y][i][j] * (1.0(REAL64)
           - hidden.activity[y][i][j]))
```

23 Initialize 22

```
SEQ i = 0 POR HIDDEN.PR.PRCCESSOR
  SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
    hidden.delta[y][i][j] := 0.0(REAL64)
```

24 Distribute weights 22

```
IF
  x = y
    Right !  Output.Weight; output.weight[y]
  x = (((y - 1) + SIZE) REM SIZE)
    Left ?  CASE Output.Weight; output.weight[y]
  TRUE
    SEQ
      Left ?  CASE Output.Weight; output.weight[y]
      Right !  Output.Weight; output.weight[y]
```

25 Calculate hidden unit error values 22

```
INT t:
SEQ
  t := (y + d) REM SIZE
  SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
    SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
      SEQ k = 0 FOR OUTPUT.PR.PROCESSOR
        hidden.delta[y][i][j] := hidden.delta[y][i][j] +
            (output.weight[t][k][i] * output.delta[t][k][j])
```

26 Send up output weights 22

```
PAR
  Down ?  CASE Output.Weight:  output.weight[((x + d) + 1) REM SIZE]
  Up !  Output.Weight; output.weight[(x + d) REM SIZE]
```

27 Distribute weights 22

```
IF
  x = y
    Right !  Output.Weight; output.weight[((y + d) + 1) REM SIZE]
  x = (((y - 1) + SIZE) REM SIZE)
    Left ?  CASE Output.Weight; output.weight[((y + d) + 1) REM SIZE]
  TRUE
    SEQ
      Left ?  CASE Output.Weight; output.weight[((y + d) + 1) REM SIZE]
      Right !  Output.Weight:  output.weight[((y + d) + 1) REM SIZE]
```

28 Send up output delta values 22

```
PAR
  Down ?  CASE Output.Delta:  output.delta[((y + d) + 1) REM SIZE]
  Up !  Output.Delta; output.delta[(y + d) REM SIZE]
```

29 Calculate hidden unit error values 22

```
INT t:
SEQ
  t := ((y - 1) + SIZE) REM SIZE
  SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
    SEQ j = 0 FOR PATTERNS.PR.PROCESSOR
      SEQ k = 0 FOR OUTPUT.PR.PROCESSOR
        hidden.delta[y][i][j] := hidden.delta[y][i][j] +
            (output.weight[t][k][i] * output.delta[t][k][j])
```

30 Propagate input unit activity 14

```
... Distribute output unit delta values  (31)
... Distribute hidden unit activities  (32)
... Calculate gradients  (33)
... Change output weights  (34)
```

31 Distribute output unit delta values 30

```
IF
  x = y
    SKIP
  TRUE
    output.delta[x] := output.delta[y]
SEQ d = 0 FOR SIZE - 1
  PAR
    Left ?  CASE Output.Delta; output.delta[(((x - d) + SIZE) - 1) REM
                                            SIZE]
    Right !  Output.Delta; output.delta[((x - d) + SIZE) REM SIZE]
```

32	**Distribute hidden unit activities**	**30**

```
IF
  x = y
    SKIP
  TRUE
    hidden.activity[x] := hidden.activity[y]
IF
  x = y
    SEQ d = 0 FOR SIZE - 1
      Left ?  CASE Hidden.Activity;
                   hidden.activity[(((y - 1) - d) + SIZE) REM SIZE]
  TRUE
    SEQ
      SEQ SEQ d = 0 FOR ((((x - y) + SIZE) - 1) REM SIZE)
        PAR
            Left ?  CASE Hidden.Activity;
                         hidden.activity[((y + 1) + d) REM SIZE]
            Right ! Hidden.Activity;
                    hidden.activity[(y + d) REM SIZE]
    Right !  Hidden.Activity; hidden.activity[((x - 1) + SIZE) REM SIZE]

IF
  x = y
    SEQ d = 0 FOR SIZE - 1
      Up !  Hidden.Activity:  hidden.activity[d]
  ((x + 1) REM SIZE) = y
    SEQ d = 0 FOR SIZE - 1
      Down ?  CASE Hidden.Activity; hidden.activity[d]
  TRUE
    SEQ
      Down ?  CASE Hidden.Activity; hidden.activity[0]
      SEQ d = 0 FOR SIZE - 1
        PAR
          Down ?  CASE Hidden.Activity:  hidden.activity[d + 1]
          Up !  Hidden.Activity:  hidden.activity[d]
    Up !  Hidden.Activity; hidden.activity[SIZE - 1]
```

33 Calculate gradients 30

```
SEQ i = 0 FOR OUTPUT.PR.PROCESSOR
  SEQ j = 0 FOR HIDDEN.PR.PROCESSOR
    output.change[i][j] := 0.0(REAL64)
SEQ d = 0 FOR SIZE
  SEQ i = 0 FOR OUTPUT.PR.PROCESSOR
    SEQ j = 0 FOR HIDDEN.PR.PROCESSOR
    SEQ k = 0 FOR PATTERNS.PR.PROCESSOR
      output.change[i][j] := output.change[i][j] +
          (output.delta[d][i][k] * hidden.activity[d][j][k])
```

34 Change output weights 30

```
SEQ i = 0 FOR OUTPUT.PR.PROSSOR
  SEQ
    output.bias.change[i] :- (learning.rate * output.bias.change[i]) +
                             (momentum * old.output.bias.change[i])
    SEQ j = 0 FOR HIDDEN.PR.PROCESSOR
      output.change[i][j] := (learning.  rate * output.change[i][j]) +
                             (momentum * old.output.change[i][j])

SEQ i = 0 FOR OUTPUT.PR.PROCESSOR
  SEQ
    output.bias[i] := output.bias[i] + output.bias.change[i]
    SEQ j = 0 FOR HIDDEN.PR.PROCESSOR
      output.weight[x][i][j] := output.weight[x][i][j]
                                + output.change[i][j]

old.output.change := output.change
old.output.bias.change := output.bias.change
```

35 Propagate input unit activity 14

```
... Distribute hidden unit delta values  (36)
... Distribute input unit activities  (37)
... Calculate gradients  (38)
... Change hidden weights  (39)
```

36	**Change output weights**	**35**

```
IF
  x = y
    SKIP
  TRUE
    hidden.delta[x] := hidden.delta[y]
SEQ d = 0 FOR SIZE - 1
  PAR
    Left ?  CASE Hidden.Delta; hidden.delta[(((x - d) + SIZE) - 1) REM
                                                    SIZE]
    Right !  Hidden.Delta; hidden.delta(((x - d) + SIZE) REM SIZE]
```

37 Distribute input unit activities 35

```
IF
  x = y
    SKIP
  TRUE
    input.activity[x] := input.activity[y]
IF
  x = y
    SEQ d = 0 FOR SIZE - 1
      Left ?  CASE Input.Activity;
                   input.activity[(((y - 1) - d) + SIZE) REM SIZE]
  TRUE
    SEQ
      SEQ SEQ d = 0 FOR ((((x - y) + SIZE) - 1) REM SIZE)
        PAR
          Left ?  CASE Input.Activity;
                       input.activity[((y + 1) + d) REM SIZE]
          Right ! Input.Activity;
                  input.activity[(y + d) REM SIZE]
    Right !  Input.Activity; input.activity[((x - 1) + SIZE) REM SIZE]
IF
  x = y
    SEQ d = 0 FOR SIZE - 1
      Up !  Input.Activity:  input.activity[d]
  ((x + 1) REM SIZE) = y
    SEQ d = 0 FOR SIZE - 1
      Down ?  CASE Input.Activity; input.activity[d]
  TRUE
    SEQ
      Down ?  CASE Input.Activity; input.activity[0]
      SEQ d = 0 FOR SIZE - 1
        PAR
          Down ?  CASE Input.Activity:  input.activity[d + 1]
          Up !  Input.Activity:  input.activity[d]
    Up !  Input.Activity; input.activity[SIZE - 1]
```

38 Calculate gradients 35

```
SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
  SEQ j = 0 FOR INPUT.PR.PROCESSOR
    hidden.change[i][j] := 0.0(REAL64)
SEQ d = 0 FOR SIZE
  SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
    SEQ j = 0 FOR INPUT.PR.PROCESSOR
    SEQ k = 0 FOR PATTERNS.PR.PROCESSOR
      hidden.change[i][j] := hidden.change[i][j] +
           (hidden.delta[d][i][k] * input.activity[d][j][k])
```

39 Change hidden weights 30

```
SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
  SEQ
    hidden.bias.change[i] := (learning.rate * hidden.bias.change[i])
                           + (momentum * old.hidden.bias.change[i])
      SEQ j = 0 FOR INPUT.PR.PROCESSOR
         hidden.change[i][j] :  = (learning.  rate * hidden.change[i][j])
                                 + (momentum * old.hidden.ehange[i][j])

SEQ i = 0 FOR HIDDEN.PR.PROCESSOR
    SEQ
      hidden.bias[i] := hidden.bias[i] + hidden.bias.change[i]
      SEQ j = 0 FOR INPUT.PR.PROCESSOR
         hidden.weight[x][i][j] := hidden.weight[x][i][j]
                                 + hidden.change[i][j]

old.hidden.change := hidden.change
old.hidden.bias.change := hidden.bias.change
```

40 Terminate 5

```
IF
  processor.number = 1
    Up !  Time
  TRUE
    SKIP
```

B.10 Net Partitioning Back-Propagation

The program given in the following is the program for the administrator (the program running on transputer 0). For the net partitioning algorithm the program used for the transputers is exactly alike (apart from the variable 'processor.number'), hence we do show the program used by the slaves.

The merging of step four and five of the algorithm as suggested in section 4.1.6 is done. This can be seen in fold number 17 and 18.

1	**Net partitioning algorithm**	0

```
...  Libraries (2)
...  Constants (3)
...  Protocol  (4)
...  PROC Administrator(In, Out)  (5)
...  Configure transputer 0  (32)
```

2	**Libraries**	1

```
#USE time
#USE linkaddr
#USE userio
#USE dblmath
#USE snglmath
```

3 Constants 1

```
VAL learning.rate  IS  0.2(REAL64):
VAL momentum       IS  0.9(REAL64):
VAL low.weight     IS -0.3(REAL64):
VAL heigh.weight   IS  0.3(REAL64):

VAL NUMBER.OF.PROCESSORS       IS  40:
VAL NUMBER.OF.ITERATIONS       IS 100:
VAL NUMBER.OF.PATTERNS         IS  10:
VAL INPUT.UNITS                IS  40:
VAL HIDDEN.UNITS               IS  40:
VAL OUTPUT.UNITS               IS  40:
VAL INPUT.UNITS.PR.PROCESSOR   IS (INPUT.UNITS + (NUMBER.OF.PROCESSORS
                                   - 1)) / NUMBER.OF.PROCESSORS:
VAL INPUT.UNITS.PR.PROCESSOR   IS (HIDDEN.UNITS + (NUMBER.OF.PROCESSORS
                                   - 1)) / NUMBER.OF.PROCESSORS:
VAL OUTPUT.UNITS.PR.PROCESSOR  IS (OUTPUT.UNITS + (NUMBER.OF.PROCESSORS
                                   - 1)) / NUMBER.OF.PROCESSORS:

VAL processor.number IS 0:     -- Administrator
```

4 Protocol 1

```
PROTOCOL Data.Protocol
  CASE
    Input.Data; [INPUT.UNITS.PR.PROCESSOR]REAL64
    Hidden.Data; [HIDDEN.UNITS.PR.PROCESSOR]REAL64
:
```

5	PROC Administrator(In, Out)	1

```
PROC Administrator(CHAN OF Data.Protocol In, Out)
  ... Variables  (6)
  ... Procedures and functions   (7)
  SEQ
     ... Initialize  (27)
    -- load.patterns ()
    create.net ()

    Timer ?  start

    ...   Pattern update   (31)

    Timer ?  stop

    write.full.string(screen, "Number of processors:  ")
    write.int(screen, NUMBER.OF.PROCESSORS, 0))
    write.full.string(screen, "*c*nTime used:  ")
    write.real64(screen, TicksToSecs (stop MINUS start), 0, 0))
    newline(screen)
:
```

6 Variables 5

```
[NUMBER.OF.PROCESSORS][INPUT.UNITS.PR.PROCESSOR]REAL64 input.unit.activity:
[NUMBER.OF.PROCESSORS][HIDDEN.UNITS.PR.PROCRSSOR]REAL64 hidden.unit.activity:
[HIDDEN.UNITS.PR.PROCESSOR]REAL64 hidden.unit.delta:
[OUTPUT.UNITS.PR.PROCESSOR]RRAL64 output.unit.activity:
[OUTPUT.UNITS.PR.PROCESSOR]REAL64 output.unit.delta:

[INPUT.UNITS+1][HIDDEN.UNITS.PR.RROCESSOR]REAL64 hidden.link.weight:
[INPUT.UNITS+1] [HIDDEN.UNITS.PR.PROCESSOR]REAL64 hidden.link.change:
[HIDDEN.UNITS+1][OUTPUT.UNITS.PR.PROCESSOR]REAL64 output.link.weight:
[HIDDEN.UNITS+1][OUTPUT.UNITS.PR.PROCESSOR]REAL64 output.link.change:

[NUMBER.OF.PATTERNS][INPUT.UNITS.PR.PROCESSOR]REAL64 input.pattern:
[NUMBER.OF.PATTERNS] [OUTPUT.UNITS.PR.PROCESSOR]REAL64 target.pattern:

INT LOCAL.INPUT.UNITS, LOCAL.HIDDEN.UNITS, LCCAL.OUTPUT.UNITS:
INT processor, start.index:
[NUMBER.OF.PROCESSORS]INT input.unit.index, input.unit.count:
[NUMBER.OF.PROCESSORS]INT hidden.unit.index, hidden.unit.count:
[NUMBER.OF.PROCESSORS]INT output.unit.index, output.unit.count:
TIMER Timer:
INT start, stop:
```

7 Procedures and functions 5

```
-- PROC load.patterns()
... PROC create.net()  (8)
... FUNCTION calculate.activity(net.input)  (11)
... PROC propagate.activity(pattern)  (12)
... PROC calculate.weight.changes(pattern)  (15)
```

8 PROC create.net() 7

```
PROC create.net()
  REAL64 ran:
  INT64 seed:
  SEQ
    ...  Initialize weights between input and hidden layer  (9)
    ...  Initialize weights between hidden and output layer  (10)
:
```

9 **Initialize weighs between input and hidden layer** **8**

```
SEQ j = 0 FOR LOCAL.HIDDEN.UNITS
  SEQ
    seed := (INT64 ((hidden.unit.index[processor.number] + j) + 1))
    SEQ i = 0 FOR INPUT.UNITS + 1
      SEQ
            ran, seed := DRAN(seed)
            hidden.link.weight[i][j] :=
              low.weight + (ran * (high.weight - low.weight))
            hidden.link.change[i][j] := 0.0(REAL64)
```

10 **Initialize weights between hidden and output layer** **8**

```
SEQ j = 0 FOR LOCAL.OUTPUT.UNITS
  SEQ
    seed := (INT64 ((output.unit.index[processor.number] + j) + 1))
    SEQ j = 0 FOR HIDDEN.UNITS + 1
      SEQ
        ran, seed := DRAN(seed)
        output.link.weight[i][j] :=
             low.weight + (ran * (high.weight - low.weight))
        output.link.change[i][j] := 0.0(REAL64)
```

11 **FUNCTION calculate.activity(net.input)** **7**

```
REAL64 FUNCTION calculate.activity(VAL REAL64 net.input)
  REAL64 result:
  VALOF
    result := 1.0(REAL64) / (1.0(REAL64) + DEXP(-net.input))
    RESULT result
:
```

12 **PROC propagate.activity(pattern)** **7**

```
PROC propagate.activity(VAL INT pattern)
  SEQ
    ...  Calculate output from hidden units (13)
    ...  Calculate output from output units (14)
:
```

13	**Calculate output from hidden units**	**12**

```
[HIDDEN.UNITS.PR.PROCESSOR]REAL64 hidden.net:
SEQ
  input.unit.activity[processor.number] := input.pattern[pattern]
  SEQ j = 0 FOR LOCAL.HIDDEN.UNITS
    hidden.net[j] := hidden.link.weight[INPUT.UNITS][j]
  SEQ p = processor.number FOR NUMBER.OF.PROCESSORS - 1
    SEQ
      processor := p REM NUMBER.OF.PROCESSORS
      PRI PAR
        PAR
          Out !  Input.Data; input.unit.activity[processor]
          In ?  CASE Input.Data; input.unit.activity[(processor + 1) REM
                                                  NUMBER.OF.PROCESSORS]
        SEQ
          start.index := input.unit.index[processor]
          SEQ j = 0 FOR LOCAL.HIDDEN.UNITS
            SEQ i = 0 FOR input.unit.count [processor]
              hidden.net[j] := hidden.net[j] +
                    (input.unit.activity[processor][i] *
                    hidden.link.weight[start.i.ndex + i][j])
  prooessor := (processor.number + (NUMBER.OF.PROCESSORS - 1)) REM
          NUMBER.OF.PROCESSORS
  start.index := input.unit.index[prooessor]
  SEQ j = 0 FOR LOCAL.HIDDEN.UNITS
    SEQ
      SEQ i = 0 FOR input.unit.count[processor]
        hidden.net[j] := hidden.net[j] +
             (input.unit.activity[processor][i] *
             hidden.link.weight[start.index + i][j])
      hidden.unit.activity[processor.number][j] :=
             calculate.activity(hidden.net[j])
```

14 Calculate output from output units 12

```
[OUTPUT.UNITS.PR.PROCESSOR]REAL64 output.net:
SEQ
  SEQ j = 0 FOR LOCAL.OUTPUT.UNITS
    output.net[j] := output.link.weight[HIDDEN.UNITS][j]
  SEQ p = processor.number FOR NUMBER.OF.PROCESSORS - 1
    SEQ
      processor := p REM NUMBER.OF.PROCESSORS
      PRI PAR
        PAR
             Out !  Hidden.Data; hidden.unit.activity[processor]
             In ?  CASE Hidden.Data; hidden.unit.activity[(processor + 1)
                                              REM NUMBER.OF.PROCESSORS]
        SEQ
             start.index := hidden.unit.index[processor]
             SEQ j = 0 FOR LOCAL.OUTPUT.UNITS
               SEQ i = 0 FOR hidden.unit.count [processor]
                 output.net[j] :=
                       output.net[j] +
                       (hidden.unit.activity[processor][i]
                       * output.link.weight[start.i.ndex + i][j])
  processor := (proaessor.number + (NUMBER.OF.PROCESSORS - 1)) REM
      NUMBER.OF.PROCESSORS
  start.index := hidden.unit.index[processor]
  SEQ j = 0 FOR LOCAL.OUTPUT.UNITS
    SEQ
      SEQ i = 0 FOR hidden.unit.count[processor]
        output.net[j] := output.net[j] + (hidden.unit.activity[processor][i]
              * output.link.weight[start.index + i][j])
      output.unit.activity[j] := calculate.activity(output.net[j])
```

15 PROC calculate.weight.changes(pattern) 7

```
PROC calculate.weight.changes(VAL INT pattern)
  SEQ
    ... Calculate output unit delta values (15)
    ... Calculate hidden unit deltas and weight changes to output units (17)
    ... Calculate changes of input to hidden unit weights (25)
    ... Change input to hidden unit weights (26)
:
```

16 Calculate output unit delta values 15

```
SEQ i = 0 FOR LOCAL.OUTPUT.UNITS)
  output.unit.delta[i] :=
     (target.pattern[pattern] [i] - output.unit.activity[i]) *
     (output.unit.activity[i] * (1.0(REAL64) - output.unit.activity[i]))
```

17 Calculate hidden unit deltas and weight changes to output units 15

```
[HIDDEN.UNITS.PR.PROCESSOR]REAL64 in.data, out.data:
SEQ
  processor := (processor.number + 1) REM NUMBER.OF.PROCESSORS
  start.index := hidden.unit.index[prooessor]
  SEQ i = 0 FOR hidden.unit.count[prooessor]
    SEQ
      out.data[i] := 0.0(REAL64)
      SEQ j = 0 FOR LOCAL.OUTPUT.UNITS
        SEQ
          out.Data[i] := out.data[i] +
            (output.unit.delta[j] * output.link.weight[start.index + i][j])
          ... Calaulate changes of hidden to output unit weights (18)
          ... Change hidden to output unit weights (13)
  ...   Loop (20)
  SEQ i = 0 FOR LOCAL.HIDDEN.UNITS
    hidden.unit.delta[i] := out.data[i] *
        (hidden.unit.activity[processor.number][i] *
        (1.0(REAL64) - hidden.unit.activity[processor.number][i]))
  SEQ j = 0 FOR LOCAL.OUTPUT.UNITS
    SEQ
      ... Calculate changes of hidden to output unit weights (23)
      ... Change hidden to output unit weights (24)
```

18 Calculate changes of hidden to output unit weights 17

```
output.link.change[start.index + i][j] := (learning.rate *
    (output.unit.delta[j] * hidden.unit.activity[processor][i])) +
    (momentum * output.link.change[start.index + i][j])
```

19 Change hidden to output unit weights 17

```
output.link.weight[start.index + i][j] :=
  output.link.weight[start.index + i][j] +
  output.link.change[start.index + i][j]
```

20 Loop 17

```
SEQ p = (processor.number + 2) FOR NUMBER.OF.PROCESSORS - 1
  SEQ
    PRI PAR
      PAR
        Out !  Hidden.Data; out.data]
        In ?  CASE Hidden.Data;in.data
      SEQ
        processor := p REM NUMBER.OF.PROCESSORS
        start.index := hidden.unit.index[processor]
        SEQ i = 0 FOR hidden.unit.oount[proaessor]
          SEQ
            hidden.unit.delta[i] := 0.0(REAL64)
            SEQ j = 0 FOR LOCAL.OUTPUT.UNITS
              SEQ
                hidden.unit.delta[i] := hidden.unit.delta[i] +
                  (output.unit.delta[j] *
                   output.link.weight[start.index + i][j])
                ... Calculate changes of hidden to output unit weights (21)
                ... Change hidden to output unit weights (22)
    SEQ i = 0 FOR hidden.unit.count[processor]
        out.data[i] := in.data[i] + hidden.unit.delta[i]
```

21 Calculate changes of hidden to output unit weights 20

```
output.link.change[start.index + i][j] := (learning.rate *
     (output.unit.delta[j] * hidden.unit.activity[processor][i])) +
     (momentum * output.link.change[start.index + i][j])
```

22 Change hidden to output unit weights 20

```
output.link.weight[start.index + i][j] :=
     output.link.weight[start.index + i][j] +
     output.link.change[start.index + i][j]
```

23 Calculate changes of hidden to output unit weights 17

```
output.link.change[HIDDEN.UNITS][j] :=
     (learning.rate * output.unit.delta[j]) +
     (momentum * output.link.change[HIDDEN.UNITS][j])
```

24 Change hidden to output unit weights 17

```
output.link.weight[HIDDEN.UNITS][j] :=
    output.link.weight[HIDDEN.UNITS][j] +
    output.link.change[HIDDEN.UNITS][j]
```

25 Calculate changes of input to hidden unit weights 15

```
SEQ p = 0 FOR NUMBER.OF.PROCESSORS
  SEQ
    start.index :  = input.unit.index[p]
    SEQ j = 0 FOR LOCAL.HIDDEN.UNITS
      SEQ i = 0 FOR input.unit.count[p]
        hidden.link.ehange[start.index + i][j] := (learning.rate *
             (hidden.unit.delta[j] * input.unit.activity[p][i])) +
             (momentum * hidden.link.change[start.index + i][j])
SEQ j = 0 FOR LOCAL.HIDDEN.UNITS
  hidden.link.change[INPUT.UNITS][j] :=
       (learning.rate * hidden.unit.delta[j]) +
       (momentum * hidden.link.change[INPUT.UNITS][j])
```

26 Change input to hidden unit weights 15

```
SEQ j = 0 FOR LOCAL.HIDDEN.UNITS
  SEQ i = 0 FOR INPUT.UNITS + 1
    hidden.link.weight[i][j] :=
         hidden.link.weight[i][j] + hidden.link.change[i][j]
```

27 Initialize 5

```
... Calculate input-unit index and length data for all processors (28)
LOCAL.INPUT.UNITS := input.unit.count[processor.number]
... Calculate hidden-unit index and length data for all processors (29)
LOCAL.HIDDEN.UNITS := hidden.unit.count[processor.number]
... Calculate output-unit index and length data for all processors (30)
LOCAL.OUTPUT.UNITS := output.unit.count[processor.number]
```

28 **Calculate input-unit index and length data for all processors** 27

```
input.unit.index[0] := 0
input.unit.count[0] := INPUT.UNITS.PR.PROCRSSOR
SEQ i = 1 FOR NUMBER.OF.PROCESSORS - 1
  SEQ
    input.unit.index[i] := input.unit.index[i - 1] +
                           input.unit.count[i - 1]
      IF
        (INPUT.UNITS REM NUMBER.OF.PROCESSORS) = 0
            input.unit.count[i] := INPUT.UNITS.PR.PROCRSSOR
        (INPUT.UNITS REM NUMBER.OF.PROCESSORS) > i
            input.unit.count[i] := INPUT.UNITS.PR.PROCRSSOR
        TRUE
            input.unit.count[i] := INPUT.UNITS.PR.PROCESSOR - 1
```

29 **Calculate hidden-unit index and length data for all processors** 27

```
hidden.unit.index[0] := 0
hidden.unit.count[0] := HIDDEN.UNITS.PR.PROCRSSOR
SEQ i = 1 FOR NUMBER.OF.PROCESSORS - 1
  SEQ
    hidden.unit.index[i] := hidden.unit.index[i - 1] +
                            hidden.unit.count[i - 1]
    IF
      (HIDDEN.UNITS REM NUMBER.OF.PROCESSORS) = 0
        hidden.unit.count[i] := HIDDEN.UNITS.PR.PROCRSSOR
      (HIDDEN.UNITS REM NUMBER.OF.PROCESSORS) > i
        hidden.unit.count[i] := HIDDEN.UNITS.PR.PROCRSSOR
      TRUE
        hidden.unit.count[i] := HIDDEN.UNITS.PR.PROCESSOR - 1
```

30 **Calculate output-unit index and length data for all processors** **27**

```
output.unit.index[0] := 0
output.unit.count[0] := OUTPUT.UNITS.PR.PROCRSSOR
SEQ i = 1 FOR NUMBER.OF.PROCESSORS - 1
  SEQ
    output.unit.index[i] := output.unit.index[i - 1] +
                                output.unit.count[i - 1]
      IF
        (OUTPUT.UNITS REM NUMBER.OF.PROCESSORS) = 0
          output.unit.count[i] := OUTPUT.UNITS.PR.PROCRSSOR
        (OUTPUT.UNITS REM NUMBER.OF.PROCESSORS) > i
          output.unit.count[i] := OUTPUT.UNITS.PR.PROCRSSOR
        TRUE
          output.unit.count[i] := OUTPUT.UNITS.PR.PROCESSOR - 1
```

31 **Pattern update** **5**

```
INT pattern:
INT32 seed:
REAL32 ran:
SEQ
  seed := 1(INT32)]
  SEQ iteration = 0 POR NUMBER.OF.ITERATIONS
    SEQ
      ran, seed := RAN(seed)
      IF
        seed < 0 (INT32)
          seed := -seed
        TRUE
          SKIP
      pattern := (INT seed)
      propagate.activity(pattern)
      caloulate.weight.changes(pattern)
```

32 **Configure transputer** **1**

```
CHAN OF Data.Protocol In, Out:
PLACE In AT Link1in:
PLACE Out AT Link3out:
Administrator(In, Out)
```

Bibliography

[Abu-Mostafa] Yaser S. Abu-Mostafa, Demetri Psaltis *Optical Neural Computers*
Scientific American. March 1987

[Blelloch] Guy Blelloch, Charles R. Rosenberg *Network Learning on the Connection Machine*
Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy 1987

[Bourrely] Jean Bourrely *Parallelization of a neural learning algorithm on a Hypercube*
Hypercube and Distributed Computers, F, André and J. P. Verjus (Editors), Elsevier Science Publishers B. V. (North-Holland), 1989

[Christiansen] L. Christiansen, H. Tolbøl *Anvendelse og evaluering af Transputer net* (*the Use and Evaluation of Transputer Nets*)
Thesis, Computer science Department, University of Aarhus, Denmark

[Ernoult] Christine Ernoult *Performance of Backpropagation on a Parallel Transputer-Based Machine*
International Workshop, Neural Nets and Their Applications, Proceedings and Exhibition Catalog, Neuro-Nimes 1988

[Forrest] B. M. Forrest, D. Roweth, N. Stroud, D. J. Wallace, G. V. Wilson *Implementing Neural Network Models on Parallel Computers*
The Computer Journal, Vol. 30, No. 5, 1987

[Fox1] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, D. W. Walker *Solving Problems On Concurrent Processors. Volume 1: General Techniques and Regular Problems, Sec. 3–6*
Prentice Hall, 19??

[Fox2] G. Fox, S. Otto, A. Hey *Matrix algorithm on a hypercube: matrix multiplication*
Parallel Computing, Vol. 4, 17 (1987)

[Hoare] C. A. R. Hore *Computing Sequential Processes*
Communications of ACM, Vol. 21, pp 666–677, 1978

[Inmos1] INMOS Limited *Transputer Development System*
Prentice Hall, 1988

[Inmos2] INMOS Bristol *Technical note 6: IMS T800 Architecture*
INMOS Limited

[Johansson] E. M. Johansson, F. U. Dowla, D. M. Goodman *Backpropagation Learning for Multi-Layer Feed-Forward Neural Networks Using the Conjugate Gradient Method*
Lawrence Livermore National Laboratory, Preprint, UCRL-JC-104850, 1990

[Millán] José del R. Millán. Pau Bofill *Learning by back-propagation: a systolic algorithm and its transputer implementation*
Neural Networks, Vol. 1. No. 3, July 1989

[Minsky] M. Minsky, S. Papert *Perceptrons*
MIT Press, Cambridge MA, 1969

[Petrowski] A. Petrowski, L. Personnaz, G. Dreyfus, C. Girault *Parallel Implementations of Neural Network Simulations*
Hypercube and Distributed Computer, F. André and J. P. Verjus (Editors), Elsvier Science Publishers B. V. (North-Holland), 1989

[Pomerleau1] Dean, A. Pomerleau, George L. Gusciora, David S. Touretzky, H. T. Kung *Neural Network simulation at Warp Speed: How We Got 17 Million Connections Per Second*

IEEE International Conference on Neural Networks, Vol. 2, 1988

[Pomerleau2] Dean A. Pomerleau *Understanding Neural Network Simulator Performance*
Neural Network Review, Vol. 4, No. 1, 1990

[Rosenblatt] F. Rosenblatt *Principles of Neurodynamics*
New York: Spartan, 1962

[Rumelhart] David E. Rumelhart, James L. McClelland, and the PDP Research Group *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*
The MIT Press, Cambridge, Massachusetts, 1986

[Schwartz] Jacob T. Schwartz *The New Connectionism: Developing Relationships Between Neuroscience and Artificial Intelligence*
Dædalus, 1988, Proceeding of the American Academy og Arts and Sciences, Vol. 117, No. 1, p. 123–141

[Sejnowski] Terrence J. Sejnowski, Charles R. Rosenberg *NETtalk: a parallel network that learns to read aloud*
The Johns Hopkins University Electrical Engineering and Computer Science Technical Report JHU/EECS-86/01
Retrieving the NETtalk data set: The data set is available by anonymous FTP from the cs.cmu.edu site in the directory /ags/cs.cmu.edu/project/connect/bench.

[Singer] Alexander Singer *Implementations of Artificail Neural Networks on the Connection Machine*
Parallel Computing, Vol. 14, No. 3, August 1990

[Tank] David E. Tank, John J. Hopfield *Collective Computation in Neuronlike Circuits*
Scientific American, December 1987

[Vogl] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, D. L. Alkon *Accelerating the Convergence of the Back-Propagation Method*
Biological Cybernetics, Vol. 59, p. 257–263

[Welch] P. H. Welch *Emulating Digital Logic using Transputer Networks*
PARLE, Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures, Eindhoven, The Netherlands, June 15–19, 1987, Proceedings

[Witbrock] Michael Witbrock, Marco Zagha *An Implementation of Backpropagation Learning on GF11, a Large SIMD Parallel Computer*
Parallel Computing, Vol. 14, No 3, August 1990

[Zhang] Xiru Zhang, Michael McKenna, Jill P. Mesirow, David L. Waltz *The Backpropagation algorithm on Grid and Hypercube Architectures*
Parallel Computing, Vol. 14, No. 3, August 1990