Chapter 7

Implementation of Backpropagation Neural Networks On Large Parallel Computers

Jim Torresen
Department of Computer Systems and Telematics
Norwegian University of Science and Technology, Norway
Email: jimtoer@idt.unit.no

Shinji Tomita
Department of Information Science
Kyoto University, Japan

7.1 Introduction

The chapter will consider the problem of mapping the backpropagation training of real neural applications onto large parallel systems. Many parallel neural training programs have been implemented, but most are tested on none or few real applications. If the programs are tested for large neural networks they will usually show better performance than what is obtainable for real neural networks, which are usually small. On parallel systems with few processing elements, many mapping schemes are able to run efficiently. However, as the number of processors increases, the problem of communication overheads and uneven load becomes more prominent. Based on the inherent degrees of parallelism in the training algorithm, it is possible to suggest an efficient mapping, which minimizes the problems. In the chapter, we propose a parallel mapping of neural training, which adapts the configuration to the neural application.

A popular way of implementing backpropagation networks on a parallel machine is to partition the training set and have the processing elements train different patterns in parallel. This method inhibits updating the weight values after each pattern presentation. Experiments reported here show a linearly increasing number of iterations needed for convergence as the weights are less frequently updated. This fact can not be omitted in the mapping strategy. Thus, we suggest a heuristic for selecting the best mapping scheme, combining all degrees of parallelism, according to what minimizes the total training time.

The mapping scheme is tested on the general purpose, parallel computer Fujitsu AP1000. This is a message passing MIMD computer with a two dimensional torus network of processing elements. The largest system contains 512 processing elements. Fujitsu offers the computer for free to registered users (a limited number of hours each month). The computer is accessed over the Internet and programmed using the C programming language.

The results indicate that the training speed can be reduced from hours to minutes if the parallel system is used instead of a single processor workstation. Moreover, the flexible mapping, which adjusts the configuration to the given neural application, may result in a training speed several hundred percent faster than a fixed mapping. This by using the general purpose computer AP1000, not a specialized neurocomputer.

7.2 Hardware for Running Neural Networks

In this section, the parallel computer used in this study for implementing the back propagation training algorithm is presented. In addition, conditions for the experiments will be discussed.

7.2.1 Fujitsu AP1000

Fujitsu AP1000 is a general purpose message passing MIMD computer with distributed memory [1]. Figure 7.1 depicts the architecture. The AP1000 processing element is called a *cell* and consists of a Sparc CPU, an FPU, a message passing unit, 128 Kbyte cache

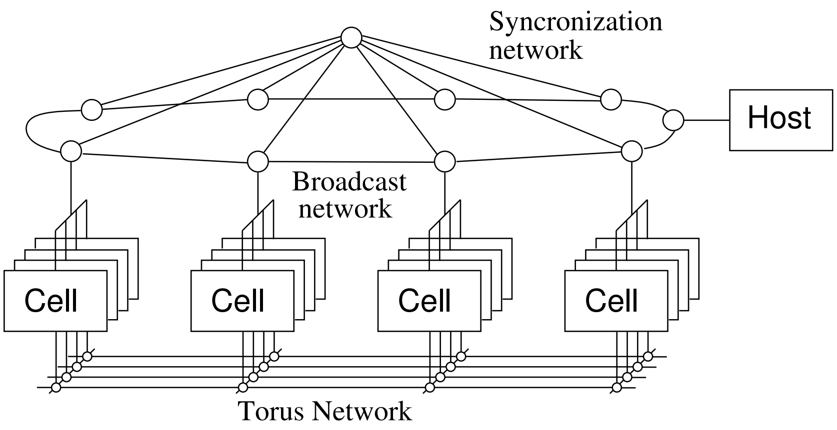

Figure 7.1: **The AP1000 architecture.**

and 16 Mbyte local memory. The largest available system consists of 512 cells[1]. The cells are connected by three different communication networks as seen in the figure. First, the cells are interconnected by bidirectional channels in a two dimensional torus network. Communication is by packet switching and packets are routed from source to destination using the wormhole routing switching method. As such, the path length has little effect on communication time. Secondly, there is a broadcast network for bidirectional communication with the host – a Sun 4 workstation running the UNIX operating system. Thirdly, there is a network for fast synchronization. The computer can be programmed using either FORTRAN or C programming languages. The languages are enhanced for doing communication etc.

The AP1000 is accessible over the Internet, without any fee, to registered users in Japan and overseas from the *Fujitsu Parallel Computing Research Facilities* (FPCRF, Kawasaki, Japan). Several versions of the computer are available: 64, 128, 256, and 512 cells.

[1]Organized as 32 rows by 16 columns of cells.

Through an *Open Use* program, each user group can reserve any of the computers for up to 20 hours each month. In this research, we mainly used a 64 cell version located at Kyoto University, Japan and obtained the results for larger systems by running the programs at FPCRF. A similar facility with a 128 cell system, is available at the Imperial College, London.

7.2.1.1 Programming the System

To use the AP1000 computer, at least two programs are required: a host program running on the host computer and one or more cell program(s). The host program configures the cells by use of a command – a call to a library function, which specifies the number of cells in the x and y dimension. Cell *program(s)* are down-loaded to all cells by a single command. The host can communicate with cells by broadcasting to all or addressing a specific cell.

A library of global reduction functions is available in the programming language to calculate the absolute maximum, absolute minimum, or the sum of data in all cells or in cells in the same column or row of the 2D-torus network. The functions operate on single variables, not arrays. A scatter/gather data function can be used to communicate data between all cells and the host. Each cell receives or sends only the portion of the data specified.

A performance analysis tool is available which shows the activity of each cell during runtime. A trace function is also available to plot the activity of cells in a selected time interval. Support for more detailed timing information is also available.

7.2.1.2 Neural Network Implementations

Only one published work is known, except for those by the authors of this chapter, on parallel neural network implementations on AP1000. Kuga et al. have implemented a neuron parallel implementation by vertical slicing of the feed-forward network [2]. Thus, each PE contains a slice of the network. Three different methods for all-to-all communication are compared. First one-to-one communication is used. Secondly, messages are rotated in horizontal and vertical rings. Thirdly, the sending PE is located in the center of the torus and messages are propagated through parallel routes to the other PEs. The methods are compared for the recall phase for the case of an equal number of neurons in each layer. Results are expressed in terms of speed-up for each of the methods. The second method, broadcasting by rotation, obtained the highest speedup. The difference in speedup for the methods increases as the number of PEs increases (256 neurons per layer are used).

7.2.2 Neural Network Applications Used in this Work

In this work realistic figures for network size are used by analysis of real world applications. The applications used in the experiments in this work are listed in Table 7.1.

NETTALK, which has been used in performance testing of both general purpose computers [8–14] and neurocomputers [15, 16], is regarded as a benchmark for comparing computers running the BP algorithm. However, some differences in the number of hidden units

Table 7.1: **BP applications used for measuring training performance.**

Application	Network size	# Training patterns
NETTALK [3]	203x120x26	5438
Character recognition (a X11 font set)	40x30x26	26
Image compression [4]	64 x 4–16 x 64	4096
Speech Recognition [5]	256x64–1024x64	4096
Sonar Target Classification [6, 7]	60 x 24 x 2	208

and weight updating frequency exist. In this work, NETTALK is used as the main benchmark. An increased variation in the number of hidden units has been included in the experiments to obtain more general results.

7.2.3 Experimental Conditions in this Work

The implementations were first debugged and tested for a small character recognition problem. The training performances were then measured using randomly generated training patterns on networks with size equal to real applications. Bias was not explicitly implemented, but assumed to be one of the input units and one of the hidden units set to "1". Due to the large cell memory, the *whole* training set can be stored in the cells, thus downloading is only needed initially. This means that the down-loading time is very short compared to the total training time and is omitted from the time measurements.

The weight matrices are updated according to the following equations [2].

$$\Delta w_{o,kj}(p+1) = \alpha \Delta w_{o,kj}(p) + (1-\alpha)\delta_{o,k}(p+1) y_{h,j}(p+1) \quad (7.1)$$

and

$$w_{o,kj}(p+1)' = w_{o,kj}(p) + \eta \Delta w_{o,kj}(p+1) \quad (7.2)$$

As the main purpose of these experiments has not been to show the computer speed, little time has been used on program code optimization.

7.3 General Mapping onto 2D-torus MIMD Computers

The topic of this section is to make a mapping of BP onto a parallel processor, which minimizes the total training time of *any* BP application. Most real implementations of BP are based on a heuristic with a fixed assignment of processors to each involved degree of parallelism. However, the different kinds of applications – neural network structure and training set, require different mappings in order to obtain minimum training time. Thus, the performance of these fixed implementations are highly application dependent. In many publications on parallel BP mappings, performance is only measured for at

[2] Same as Equations 1.16 and 1.17 of Chapter 1

most one real application. For instance, some select a non realistic network, usually large, that runs near optimally on the parallel machine, without reporting the performance of real applications. Also, when training set parallelism is used, the weight update frequency is usually not considered.

In this section a general mapping scheme, which includes all degrees of parallelism, is proposed. It is a heuristic for mapping feed-forward neural networks near optimally onto a 2D-torus MIMD computer. The assignment of cells to each degree of parallelism is done according to the given neural network and training set. That is, the method selects the appropriate mapping according to the given application. The method will run on any number of processors, available in todays parallel computers. However, the mapping's main advantage is for the case of a large number of processors. Small networks and training sets in particular can run more efficiently on larger parallel systems.

Today's technology allows more logic to be placed on a given circuit area. It is therefore expected that tomorrows technology will allow a larger number of processors on a single circuit and that parallel systems will consist a of larger number of processors. Massively parallel computers will be more available than the few presently available machines. These massively parallel systems can overcome the limit of sequential program execution by several orders of magnitude if the parallel algorithms can scale efficiently.

The main interest here is to minimize the *total* training time. Thus, it is impossible to omit the selection of a proper weight update interval when training set parallelism is used. Few of the published mapping methods address the issue of total training time, which is a topic of this study. To obtain high speedup on a highly parallel computer, it was shown in the previous work that the issue is to combine *multiple* degrees of parallelism [17]. While the previous work was on fixed combinations of the degrees of parallelism, the new scheme allows arbitrary combinations of training set parallelism, pipelining and node parallelism.

Some mapping strategies (e.g. [13,18]) are based on assigning one processing element to each neuron. In this way, the utilization of the parallel computer depends on the neural network. This assignment requires a certain number of available processors and there is a possibility of processors being idle. A better approach is to base the mapping on the number of available processors and then ask how the different degrees should be combined to in the best way utilize the available processors. Thus, to combine the different parallel BP training dimensionalities in the way that minimizes the total training time. This is the underlying idea of the mapping that will be presented below. The name *application adaptable* mapping is given to this scheme. Parts of this work is published in [19,20].

7.3.1 The Proposed Mapping Scheme

The proposed mapping aim at obtaining the best possible load balance between the processing cells for *any* neural network application. Hence, the number of processors assigned for each degree of parallelism is made to be changeable. The mapping will be presented first in a version without pipelining and secondly in a version including pipelining. Further, some improvements will be given.

The first part of the proposed mapping is shown in Figure 7.2 and

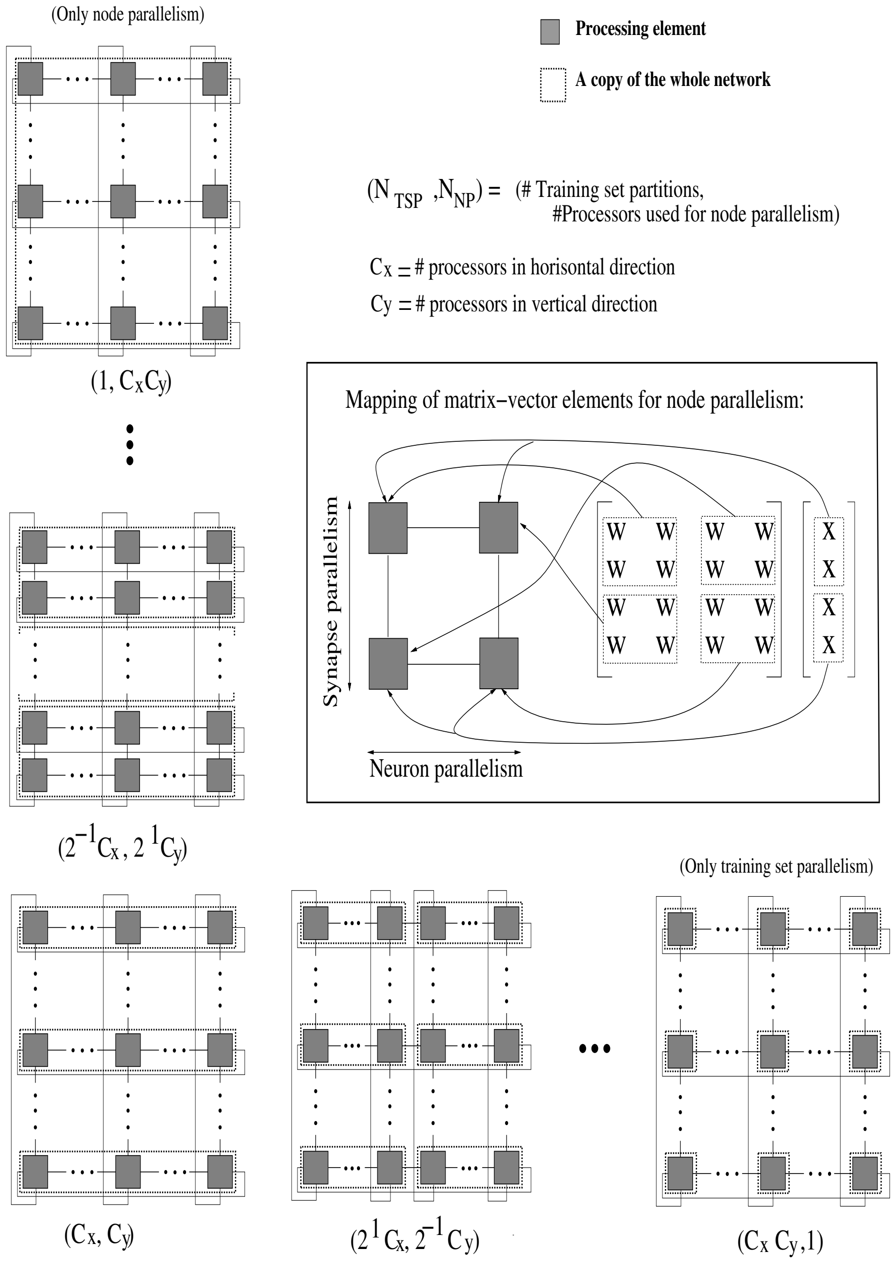

Figure 7.2: **BP mappings with training set and node parallelism.**

consists of a set of possible system configurations. Each configuration combines node and training set parallelism. Within each dotted rectangle node parallelism is used. The box within the figure indicates the mapping of the elements for node parallelism. Neuron parallelism is represented within each row of processors. Synapse parallelism is achieved in the vertical dimension. This scheme is based on an ordinary parallelization of matrix-vector multiplication, as described in Section 2.2.2.3 [21]. The lower part of the figure gives a selection of the number of training set partitions versus processors assigned to neuron parallelism. Correspondingly in the left-hand part of the figure, the choice in ratio between training set parallelism and synapse parallelism is shown. This illustrates the flexibility to adjust the parallel configuration to both the neural network size and training set size.

The mapping can be extended by including pipelining, i.e., make one processing element box in the figure into one O–H pair as shown in Figure 7.3. The processors marked **H** compute the hidden layer part, while the output layer part is computed by the processors marked **O**. This scheme combines all degrees of BP parallelism except training session parallelism. The efficiency of including pipelining depends on the number of processors and the number of neurons in each layer. This was illustrated in [17].

Below each processor configuration in the figures, the number of training set partitions N_{TSP} and the number of processors for node parallelism N_{NP} are given – (N_{TSP}, N_{NP}). For Figure 7.2 the relation between them can be expressed by

$$N_{TSP} = \frac{C_x C_y}{N_{NP}} \tag{7.3}$$

where C_x and C_y are the number of cells in the horizontal and ver-

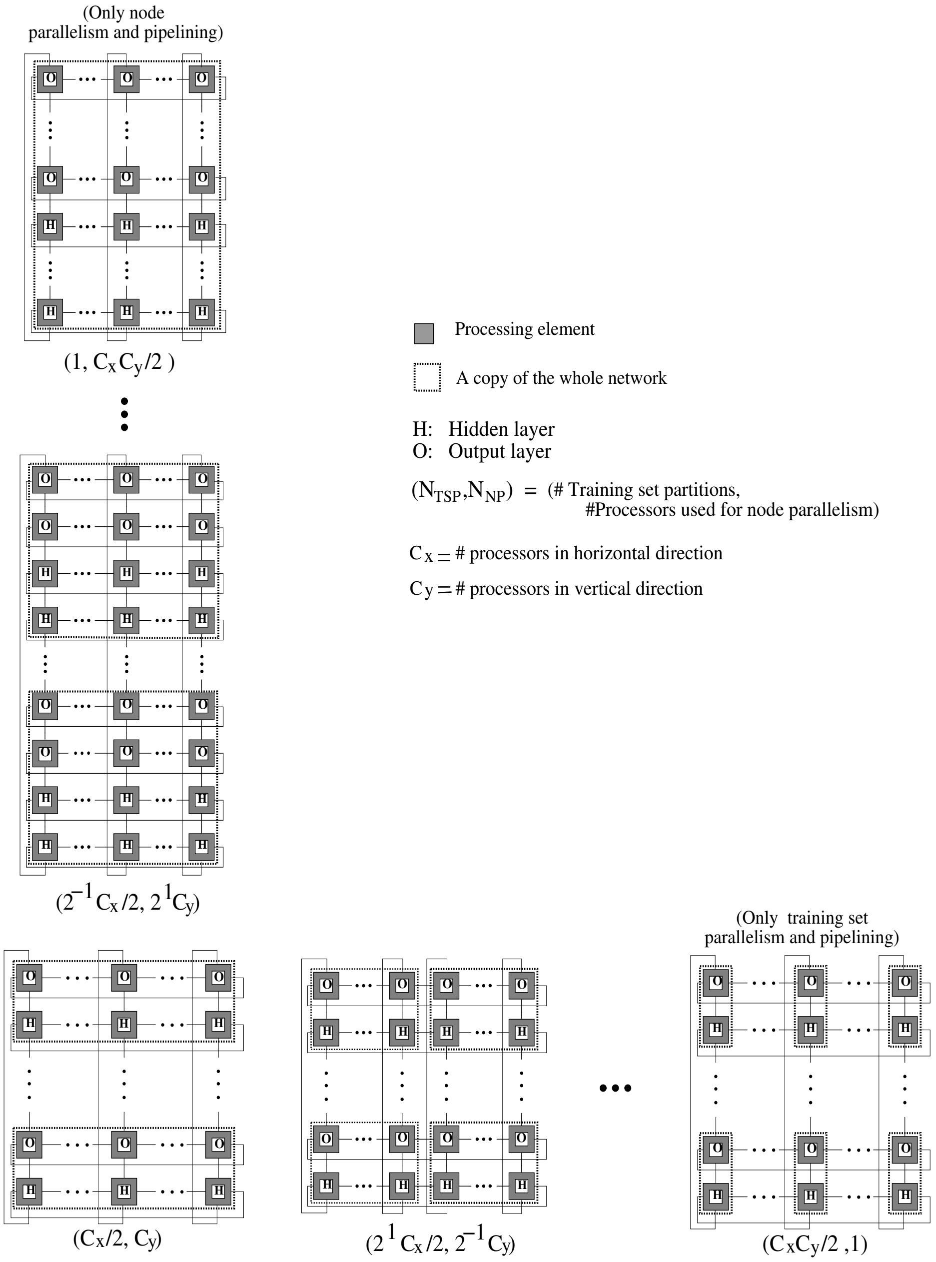

Figure 7.3: **BP mappings including training set parallelism, pipelining and node parallelism. H indicate processors training the hidden layer, while O indicate processors training the output layer.**

tical dimension respectively in the 2D-torus array[3]. If pipelining is included we get

$$N_{TSP} = \frac{C_x C_y}{2N_{NP}} \tag{7.4}$$

For a given weight update interval, a larger number of training set partitions imply fewer training pattern computations on each processor between weight updates.

One problem of the mapping scheme is network contention, i.e. several cells send messages concurrently to the same link. However, the communication time on AP1000 is reasonable also when network contention exists. It is approximately doubled if five messages are on the same channel, indicating that much time is spent within each cell for communication overhead and less on the physical transfer of data [22].

Optimizing the Weight Update

As described earlier, convergence rate improves when weights are updated frequently and as such it is important to minimize the weight update time. To avoid weight updating becoming a bottleneck in large systems, a ($\log_2 C_y$ - 1) step summing technique is proposed for summing weight change matrices in the vertical dimension. This method requires $C_y \in \{4, 8, 16, 32\}$ for AP1000. The summing algorithm is illustrated in Figure 7.4a) [23].

The summing starts in the leftmost column in the figure. Hidden layer cells send their matrices to cells south of themselves, while output layer cells send to cells to their north. The receiving cell adds its

[3]For the largest available AP1000 with 512 cells, $C_x = 16$ and $C_y = 32$

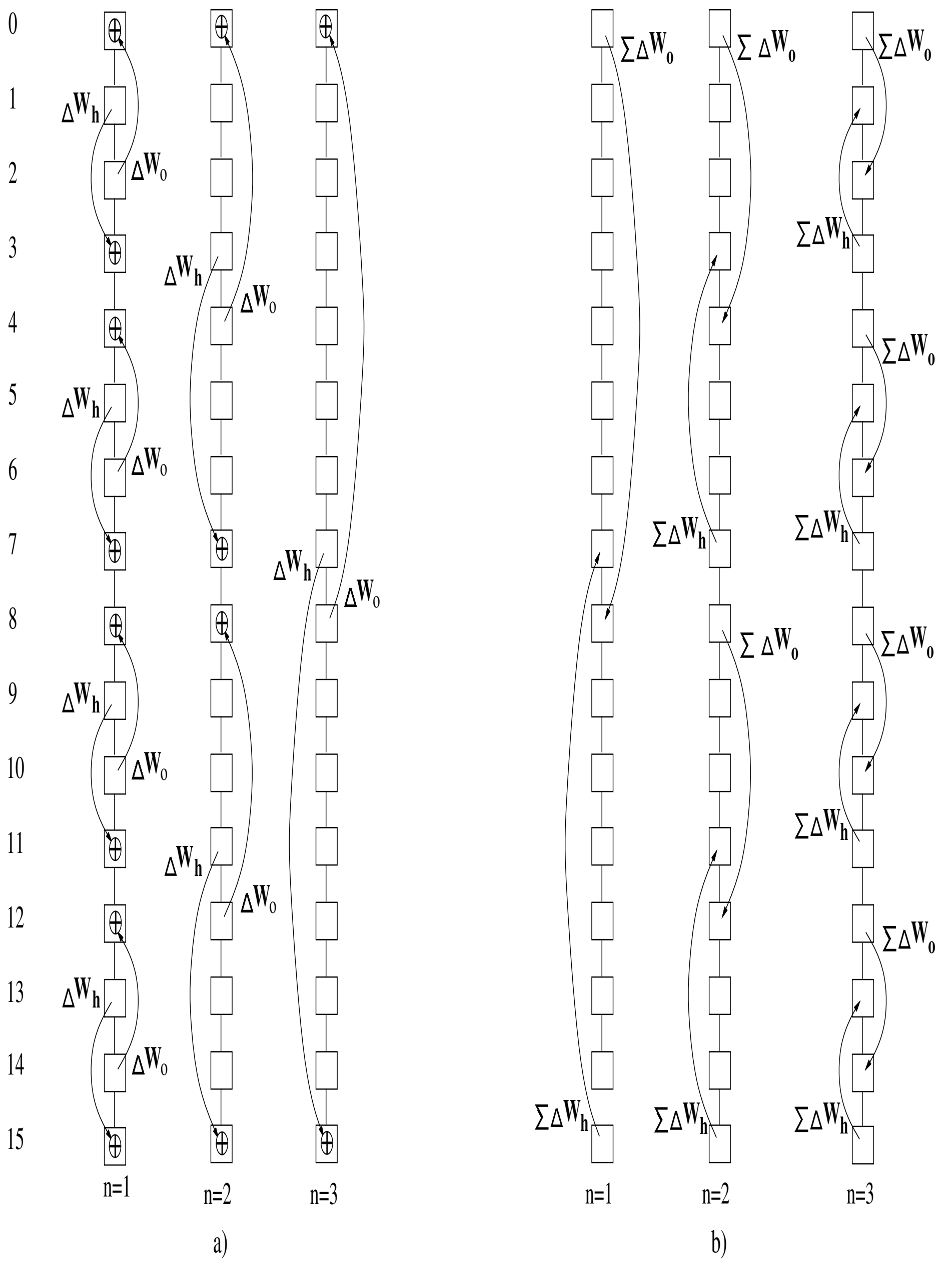

Figure 7.4: **a) Summing the weight change matrices before weight update in a system of 16 vertical cells. b) Broadcasting the summed weight change matrices back to each cell.**

local weight change matrix to the received matrix and then forwards the summed matrix. The solution is called *edge summing*, since the matrices are summed in the north-most cell and south-most cell for the output and hidden layer, respectively. In the given system, only 3 steps are required for summing the 8 weight change matrices. Then, the summed matrices are broadcasted, in the opposite direction of summing, to the cells. After each cell receives the summed weight matrix, they *update* their local weight matrices.

The same summing method can be used for the mapping without pipelining [24]. The number of summing steps becomes ($\log_2 C_y$). Thus, one more step of summing is needed. However, for the pipelining method there are additional overheads incurred due to filling the pipeline after new weights are computed.

7.3.1.1 The Implemented Mapping

To be able to evaluate the benefit of a parallel back propagation algorithm that can vary the amount of each degree of parallelism, the lower part of Figure 7.2 – configuration (C_x, C_y) to $(C_xC_y, 1)$, was implemented on AP1000. Hence, the system can be configured to an even amount of neuron and training set parallelism, just training set parallelism, or an *intermediate* configuration. The latter implies that there will be conflicts in the communication.

Comparison of the configurations (C_x, C_y) and $(C_xC_y, 1)$ has been undertaken in earlier work [17]. These are popular methods for parallelizing BP training. However, the training speed will probably be limited compared to that obtainable by using a flexible configuration.

BackPropagation_GeneralMap()
begin
$n_h = \left\lceil \frac{N_h}{N_{NP}} \right\rceil$;
$n_o = \left\lceil \frac{N_o}{N_{NP}} \right\rceil$;
$\mu_c = \left\lfloor \frac{\mu}{N_{TSP}} \right\rfloor$; { Weight update interval }
LookUp(MyCellId_x MyCellId_y);
Read learning parameters from the host;
Initialize weights;
if (MyCellId_y $\left\lfloor \frac{C_x}{N_{NP}} \right\rfloor + \left\lfloor \frac{\text{MyCellId\_x}}{N_{NP}} \right\rfloor) < (P \textbf{ mod } N_{TSP})$ **then**
MaxPattern= $\left\lfloor \frac{P}{N_{TSP}} \right\rfloor + 1$;
else
MaxPattern= $\left\lfloor \frac{P}{N_{TSP}} \right\rfloor$;
Read training patterns from the host;
while *trainend* $= 0$ **do begin**
totalerror= 0;
for $p = 1$ **to** MaxPattern **do begin**
{ Forward phase }
Compute n_h hidden outputs, $\mathbf{y}_h$;
Multiply $\mathbf{y}_h$ by the corresponding weights $\mathbf{W}_o$
for $\mathbf{y}'_o$ partial output values;
for $c = 1$ **to** C_x - 1 **do**
Send_receive_multiply_add(n_h hidden elements);
Compute sigmoid function for the n_o outputs, $\mathbf{y}_o$;
{ Backward phase }
Compute n_o elements of delta error δ_o and sum of *error*
from the n_o outputs;
error= SumError(x-dimension, *error*);
{ Global reduction function }
if error $> e$ **then**

totalerror= totalerror +1;
Compute N_h partial δ_h values; { Hidden delta error }
for c= 1 **to** C_x - 1 **do**
Send_receive_add(n_h elements of δ_h);
Accumulate_weight_changes;
if (p **mod** μ_c) = 0 **then**
Update_weights;
end; SetStatus(*totalerror* > 0);
Update_weights;
if GetStatus = 0 **then**
trainend= 1;
end;
end;

Figure 7.5: **Parallel BP training algorithm running in each cell, combining training set parallelism and neuron parallelism.**

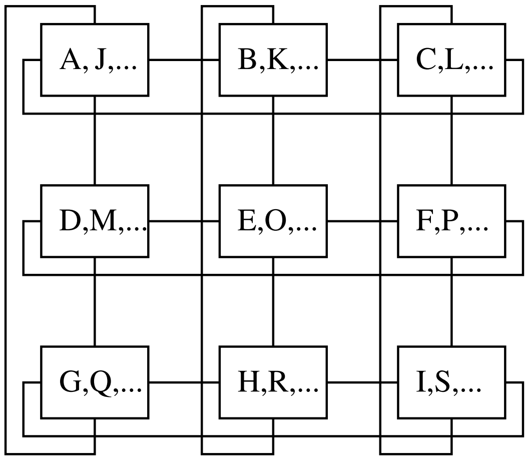

Figure 7.6: **Distribution of training pattern A, B, C,. . ..**

Pseudo-code for the implementation is given in Figure 7.5. The distribution of the training patterns is as indicated in Figure 7.6. The number of training vectors assigned to each network partition is calculated as shown in Figure 7.5 and stored in the variable MaxPattern.

7.3.1.2 Improvements of the Mapping

According to the survey of neural applications in Section 2.2.3, the neural network is often irregular in number of neurons in each layer. Further, the number of neurons in the output layer is often small. In this section, it is suggested how the mapping can be further improved with regard to these circumstances.

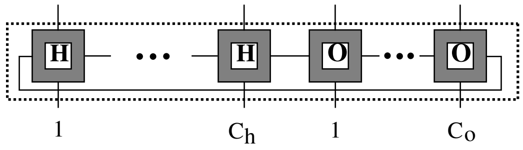

Figure 7.7: **Pipelined mapping of BP with flexible assignment of cells to each layer.**

Few Output Units To train a network efficiently when it consists of only a few output layer units, the number of processors assigned to the output layer computation should be reduced.

Figure 7.7 shows a possible pipelined mapping. This scheme can be combined with the one presented in Figure 7.2 to include pipelining. The dotted rectangle in Figure 7.7 represents one dotted rectangle in Figure 7.2. For most cases, the number of processors assigned to the hidden layer, C_h, will be larger than the number of processors assigned to the output layer, C_o. The best configuration can be found by measuring the time for one pass through the training set for different processor assignments. This scheme has been tested on the RENNS neurocomputer [25].

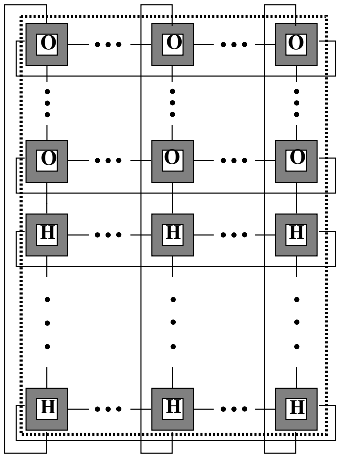

Figure 7.8: **An extension of the mapping in Figure 7.3, where the number of cells used for synapse parallelism for each layer is assigned according to the computation load.**

Improving Load Balance The scheme in the left-hand part of Figure 7.3 can be improved on load balance, as depicted in Figure 7.8. A larger number of rows is assigned (synapse parallelism) to the hidden layer with respect to the output layer. This scheme maps *large* feed-forward networks efficiently, while *small* networks should be trained by the method presented in the previous section. Both methods make the implementation adjustable to the computation load in the different layers.

7.3.1.3 Training Session Parallel Scheme

The convergence of BP is highly dependent on the various parameters like learning rate, initial weights, and number of hidden units. Moreover, hierarchical network may need training of several different networks. Thus, a training session parallel scheme can be an interesting alternative, where networks train in parallel with different initial training parameters. In Figure 7.9 a combination of training session parallelism and neuron parallelism is proposed.

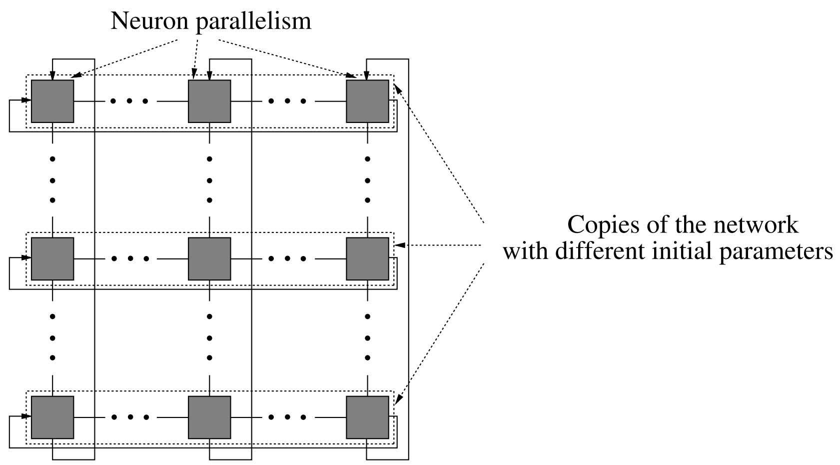

Figure 7.9: **Training session parallelism and neuron parallelism combined.**

This approach can train using *learning by pattern*. The method can be implemented as an iterative improvement scheme, where results from using different learning parameters are compared. The best ones are selected and combined for the next training session.

7.3.2 Heuristic for Selection of the Best Mapping

Before running a neural application, the mapping giving the shortest possible total training time has to be selected. First, the best initial training parameters – weights, learning rate, etc., have to be found. Then, for each mapping the total training time for a set of weight update intervals has to be predicted. The total training time is given by

$$T_{total}(\mu) = T_{1it}(\mu)N(\mu) \tag{7.5}$$

where $T_{1it}(\mu)$ is the time for one training iteration and $N(\mu)$ is the number of iterations needed for convergence, given a weight update interval μ. T_{1it} can be found either by estimation or by running each of the possible mappings for one training iteration. $N(\mu)$ can be estimated in several different ways, although a general method is not yet derived. The mapping giving the smallest possible T_{total} is selected and the corresponding μ-value is used.

1. Find optimal initial weights, number of hidden neurons, learning rate and momentum value.

2. For the given neural network, training set and computer architecture, estimate the performance for 1 iteration using the algorithm *Best_Map()*. Find the best mapping $M[\mu_i]$ for each of the weight update intervals $\mu_1, \mu_2, \ldots, \mu_n$, and store the time for 1 iteration, $T_{1it}[\mu_i]$.

   Best_Map()
   begin
   for $\mu = \mu_1, \ldots, \mu_n$ **do**
       Min_time_1it[μ] = ∞;

```
{ Pipeline not included/included }
for pipeline= 0 to 1 do
    { Examine each mapping }
    for config= 0 to (log2 (Cx · Cy) − pipeline) do
    begin
        nodepar= 2^config;
        { From Equation (7.3) and (7.4) }
        trainpar= (Cx · Cy) div (nodepar · 2^pipeline);
        for μ = μ1, ..., μn do begin
            time = Estimate_time_1_iteration(pipeline,nodepar,
            trainpar,μ);
            { Perf. impr. by reducing no of cells? }
            while (time can be reduced) do
                try reducing number of cells, update time;
            if (time < Min_time_1it[μ]) then
                { Store new map. as the best }
                update_M(pipeline,nodepar,trainpar,μ);
        end;
    end;
end;
```

3. Estimate the number of iterations needed, $N[\mu_i]$, for $\mu_1, \mu_2, \ldots, \mu_n$ equal to those in step 2.

4. Pick μ_m so that

$$T_{total}[\mu_m] = T_{1it}[\mu_m] N[\mu_m] \quad m = 1, \ldots, n$$

becomes minimized.

5. Configure the system by $M[\mu_m]$ and update the weights for every μ_m pattern.

Figure 7.10: **Heuristic in pseudo-code for selecting the best parallel mapping of BP onto a 2D-torus computer.**

The heuristic is formulated in pseudo-code in Figure 7.10. It covers all steps to make a near optimal mapping of BP onto a 2D-torus computer. It evaluates all possible mappings and picks the best one for a set of μ-values. First, some initialization of the parameters has to be done. In the second step, the performance of the mappings in Figure 7.2 and 7.3 is estimated for some given weight update intervals. The *best* configuration for each given weight update interval is found. In step 3, the number of training iterations needed is estimated for the given weight update intervals. Combining the results of steps 2 and 3 gives us in step 4 the best weight update interval and the corresponding best configuration. That is, the total training time is minimized. The heuristic is described in more detail in the next section.

The heuristic is based on the one given in [26] for selecting the best weight update interval according to the total training time. However, the mapping used is a fixed implementation based on training set parallelism. Also, experimental rules on convergence rates are deduced from a small application using synthetic data.

7.3.2.1 Selection of Back Propagation Parameters

Several papers have shown that back propagation parameter setting is of major importance to improve the convergence rate. The number of hidden neurons, initial weight values, the learning rate (η), a momentum term (α) and the weight update interval all influence the convergence rate. Both Sato [27] and Higashino [28] found from experiments that the number of training iterations N is proportional to $(1-\alpha)/\eta$ within the region in which a stable convergence is obtained. Thus, it is desirable that η takes as large a value as possible and α is as near to 1 as possible. The convergence rate can be accelerated by adaptively adjusting η and α during training as shown in [29–31].

One of the problems of back propagation is that it has a tendency to get trapped in the local minima. Different approaches have been suggested to overcome this problem. One is to use Genetic Algorithms (GA), which search a large parameter space. However, this means longer training time. GA has been used in experiments for improving BP in several ways:

- Avoid local minima by adjusting the weights by GA, either initially [32] or during training [33].
- Adjust learning rate and momentum term during training [33].
- Find the optimal neuron interconnections by adding and deleting connections between neurons in the same and different layers [34].

Since the topic of initial parameters is covered in many papers, it is not included in this work. Still, the learning parameters are very important for the convergence rate. For each applications used in this research, the best parameters have been searched for.

7.3.2.2 Estimation of Execution Time

The execution time can be estimated based on the count of floating point instructions and time for communication. A similar approach is used in [14] to estimate neural network performance on CM-2. The accuracy of the method is dependent on various aspects as idle time and cache misses. However if the accuracy is acceptable, the method of estimation is to be preferred instead of measuring real execution time, because less computation is needed.

The *Best_Map()* algorithm in Figure 7.10 examines all the possible mappings described. First, the performance using all cells available is estimated. For large systems, using fewer cells may give better performance. Thus, a test on this issue has been included. The *Estimate_time_1_iteration()*-function will be explained below.

Timing Data In Table 7.2, the floating point performance and communication timing of the AP1000 is presented [22]. The given timing is for basic operations like adding floating point variables. In a real program, overhead will be added for additional instructions for example for addressing. Thus, when estimating the performance based on the number of floating point operations this overhead is omitted. The compiler is assumed to generate code that optimally utilizes the arithmetic units in the processor. However, the result can still be of interest to estimate the theoretically maximum obtainable performance.

In the result section (Section 7.4.1.2), average timing for floating point operations measured in the real program will be used for estimation. In addition to predicting execution time, the estimation method makes it possible to determine the optimal number of processors. This is probably much larger than the number of PEs in any of the available systems.

Table 7.2: **Timing on AP1000 (ns) including loading from and storing to cache [22].**

t_{add}	442	Time to add floats sequentially.
t_{mul}	442	Time to multiply floats sequentially.
$t_{mul+add}$	646	Time for alternating multiply and add of floats.
$t_{load/store}$	880	Time for load/store when cache miss occurs.
$t_{sigmoid}$	7500	Time to compute $1/(1+e^{-x})$.
$t_{comm}(B)$	$16700 + 175B$	Communication time for sending B bytes between cells.
$t_g(C)$	$6000 + 16000 \log_2(C)$	Time for global reduction function (e.g. *x_fsum*) for C cells.
$t_{cont}(B,n)$	$T_{cont}(B,n)$	Network contention; Communication time when n messages of size B are sent between cells. See [22].

Time Estimation Below, the time used for one iteration will be estimated for the implementation described in Section 7.3.1.1. First the basic time units are computed.

Time to compute output of the hidden layer

$$T_{Fh} = n_h(N_i t_{mul+add} + t_{sigmoid}) \tag{7.6}$$

Time to compute output of the output layer

$$T_{Fo} = n_h n_o t_{mul+add} + (N_{NP} - 1)(t_{comm}(n_h) + n_h n_o t_{mul+add}) + n_o t_{sigmoid} \tag{7.7}$$

Time to compute the output layer error

$$T_{Boe} = n_o(t_{add} + t_{mul+add} + t_{mul} + t_{mul+add}) + (N_{NP} - 1)(t_{comm}(1) + t_{add}) \tag{7.8}$$

Time to compute the hidden layer error

$$T_{Bhe} = N_h n_o t_{mul+add} + (N_{NP}-1)(t_{comm}(n_h) + n_h t_{add}) + n_h(t_{mul+add} + t_{mul}) \tag{7.9}$$

Time to accumulate the output layer weight change

$$T_{Woa} = N_h n_o t_{mul+add} \tag{7.10}$$

Time to accumulate the hidden layer weight change

$$T_{Wha} = N_i n_h t_{mul+add} \tag{7.11}$$

Time to update the hidden and output layer weights

$$\begin{aligned} T_{Wupd} \;=\; & \log_2(C_y)(max\{t_{comm}(N_i n_h), t_{comm}(N_h n_o)\} && (7.12) \\ & +max\{N_i n_h t_{add}, N_h n_o t_{add}\}) + t_{comm}(N_i n_h) + t_{comm}(N_h n_o) && (7.13) \\ & +k(\frac{C_x}{N_{NP}} - 1)(t_{comm}(N_i n_h) + N_i n_h t_{add} && \\ & +t_{comm}(N_h n_o) + N_h n_o t_{add}) && (7.14) \\ & +3N_i n_h t_{mul+add} + 3N_h n_o t_{mul+add} && (7.15) \end{aligned}$$

Expressions 7.12 and 7.13 represent the vertical summing of the cell array, followed by the horizontal summing in Expression 7.14. The constant k compensates for the additional overhead, due to communication conflicts (contention). Its value will be given in the result section. The final Expression 7.15 adds the weight update time. Except for T_{Wupd}, the units are computed for *one* training pattern.

Based on these units it is possible to get an estimate of the time needed for one whole training iteration

$$T_{it}(\mu) = P_c(T_{Fh} + T_{Fo} + T_{Boe} + T_{Bhe} + T_{Wha} + T_{Woa}) + \left\lceil \frac{P}{\mu} \right\rceil T_{Wupd}$$

where P is the total number of training patterns and P_c the number of patterns in each cell.

7.3.2.3 Convergence Estimation

One of the major problems of BP training is to decrease the number of iterations needed to train an application. Much work is published on this topic. However, estimation for training time is seldom reported.

At least three approaches are possible. First, to use existing formulas [26,27] for estimation of $N(\mu)$. These are based on specific applications and compute the number of iterations needed based on the learning rate. Since these have been tested only for a few application, they may be inaccurate in general. A bound on the probabilistic rate of convergence of feed-forward networks trained to estimate a smooth function is detailed in [35].

A second approach is outlined in [20] and explained in the result section. It is based on running the algorithm for a few iterations – for various weight update intervals, and based on the initial error, the total number of iterations $N(\mu)$ used for convergence is estimated. A formula is derived for the convergence of NETTALK, based on the convergence after 25 initial iterations. By estimating the convergence based on the initial error, any techniques for improving the convergence rate is reflected in the estimation.

A third method, to be detailed later in this chapter, is to use the average ratio of convergence iterations between *learning by pattern* and *learning by epoch.*

In the result section, the convergence of several applications will be given. The different convergence estimation methods will be compared. Furthermore, some sensitivity and convergence aspects will be studied.

7.3.2.4 Modeling the Memory Usage

In this section, general expressions for the memory needed for the implemented mapping is derived. The memory allocated in each cell for the implementation described in Section 7.3.1.1 is given by

Weight matrices:	$M_w = 4 \cdot 4(N_i n_h + N_h n_o)$
	$= 16(N_i \lceil \frac{N_h}{N_{NP}} \rceil + N_h \lceil \frac{N_o}{N_{NP}} \rceil)$ byte
Training patterns:	$M_p = 4P_c(N_i + n_o)$
	$= 4\lceil \frac{P}{N_{TSP}} \rceil (N_i + \lceil \frac{N_o}{N_{NP}} \rceil)$ byte
Array and data variables:	$M_d < 10$ Kbyte
Program memory:	$M_p = 250$ Kbyte

The factor of 4 in the expressions is due to floating point variables requiring 4 bytes of memory. The expressions are given as maximum values. In the case of non-even distribution of the neurons to cells some cells may allocate less memory. These expressions can be used to show how memory requirement for each cell is reduced for an increasing number of cells.

7.3.3 Summary

This section has described a general mapping scheme, which includes all the inherent parallel degrees of the backpropagation algorithm. The mapping is flexible and combines the different degrees so that the total training time can be minimized.

To select the best implementation, a heuristic is given. This estimates the training time for the possible configurations. These estimates are used together with convergence estimates for different weight update intervals to select the best configuration and weight update interval.

In the next section, a subset of the mapping scheme will be tested. Moreover, the estimation of execution time and convergence of real neural applications will be studied.

7.4 Results on the General BP Mapping

In this section, one part of the mapping proposed in the previous section is evaluated through experiments on AP1000. The implementation is described in Section 7.3.1.1. The applications NETTALK, sonar classification, speech recognition, and image compression are used in the experiments. Training speed for different network sizes and processor configurations are given. For the two first applications the convergence for different weight update frequencies are studied. The mapping that minimizes the total training time for the parallel program is determined for NETTALK. Results from estimation of execution time will also be given.

7.4.1 NETTALK

In this section the NETTALK neural network application is used in the experiments. The section consists of four parts. First, the training speed is evaluated from measurements on the AP1000. Secondly, the performance is estimated, and thirdly, the convergence is tested and analyzed. Finally, the minimum total training time is determined. The training set consists of 1000 of the most common English words (a total of 5438 characters). Experiments have been conducted for $\mu = 63, 259, 494, 906, 1360, 2719, 5438$. The best performance reported by Sejnowski and Rosenberg [3] were by using 120 hidden units, and as such, is chosen for the experiments in this section. Bias was implemented using one of the idle input units (to be used for punctuation and word boundaries for continuous speech) and one of the 120 hidden units.

7.4.1.1 Training Speed

Figure 7.11 illustrates the performance of the general BP implementation running the NETTALK application on 64 processors. The figure makes a comparison between the training speed for 4 different weight update intervals. For each of them four different configurations are used. It is seen from the figure that it is more important to select the best configuration for small weight update intervals, than for large intervals.

The combined solutions achieve better performance than using only training set parallelism — $N_{NP} = 1$. However, for large weight update intervals e.g. $\mu = 906$ and $\mu = 5438$, smaller N_{NP} values (and thus larger N_{TSP} values) result in performance improvements. This is as expected since infrequent weight updates reduce the total weight update time for an iteration. Thus, a larger number of training set partitions can train efficiently and the most efficient configurations are those involving computer network contention — $N_{NP} < 8$. The contention free mapping — $N_{NP} = 8$, is the most efficient for frequent

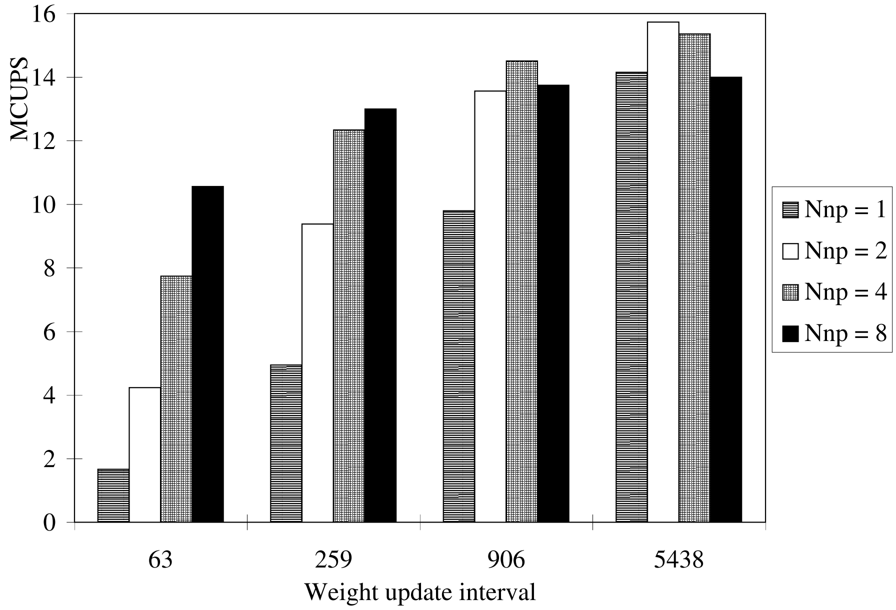

Figure 7.11: **MCUPS performance for different combinations of neuron and training set parallelism running the NETTALK application with 120 hidden neurons on a 8 x 8 cell configuration. N_{NP} = 1, 2, 4, and 8.**

weight updates.

Results for the 256 cell system are displayed in Figure 7.12. A similar performance as in the 64 cell system is seen, but the weight update interval becomes more significant. Thus, as the number of processors increases, the difference in the fastest training speed for frequent and infrequent weight updates increases. The best performance for $\mu = 5438$ is more than twice the performance for $\mu = 63$. For the largest available system consisting of 512 processing elements – see Figure 7.13, this difference is almost three. These results are understandable, since the number of training set partitions is large

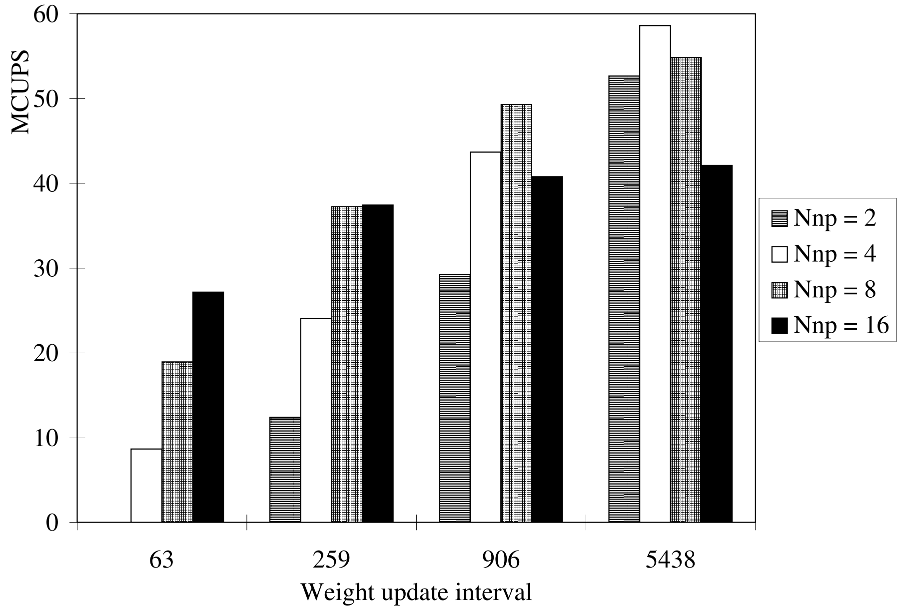

Figure 7.12: **MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 16 cell configuration. N_{NP} = 2, 4, 8, and 16.**

on a system of many processors. Thus, less training patterns are trained between weight updates. Further, more processors are assigned to each partition leading to shorter computation time for each training pattern. Hence, the impact of the weight update becomes more dominant.

A summary of the results above are plotted in Figure 7.14, where the maximum performance for 64, 256, and 512 processors is given. The performance on one cell is less than 1 MCUPS, thus the performance scaling is reasonable. This is especially true for large weight update intervals. For the largest system the best performance for μ = 906 is 86 MCUPS. This NETTALK training speed is faster than

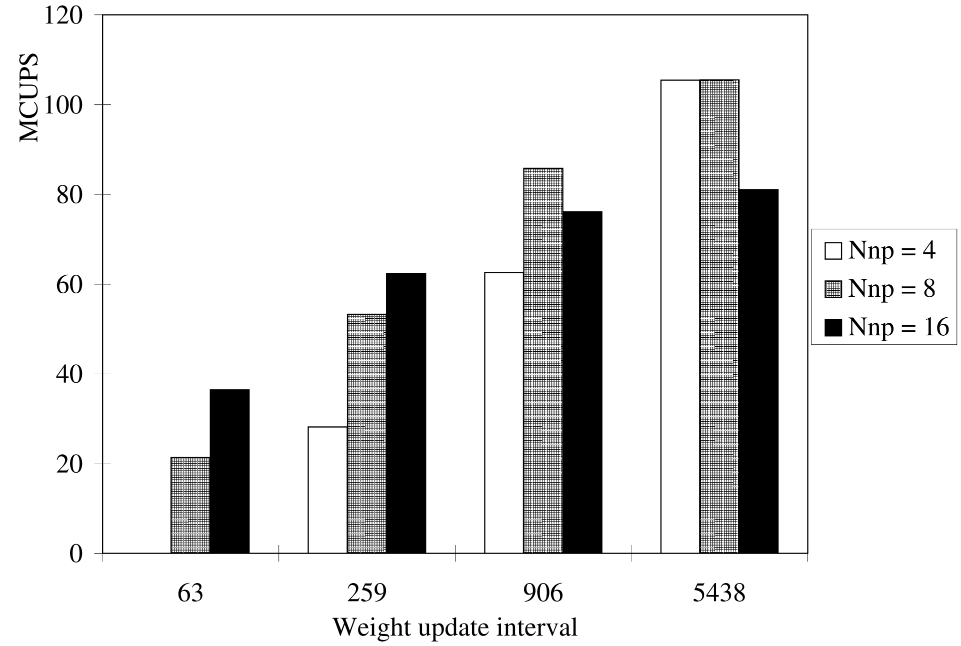

Figure 7.13: **MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 32 cell configuration. N_{NP} = 4, 8, and 16.**

those reported by others – see Table 2.5. Since the convergence is not included, other methods may require less total training time.

In Figure 7.15, the performance of the best configuration is compared to the system with the largest N_{NP} value – equal to the (C_x, C_y) configuration. The measure of the speedup between the two configurations is given, i.e. the ratio between CUPS performances. For $\mu = 63$ and $\mu = 259$ the (C_x, C_y) configuration is the fastest, and thus there is no speedup. For larger μ the speedup reaches 1.4 in the best case. This means that the performance of the best mapping is 40% faster than that of the equal degree of parallelism based

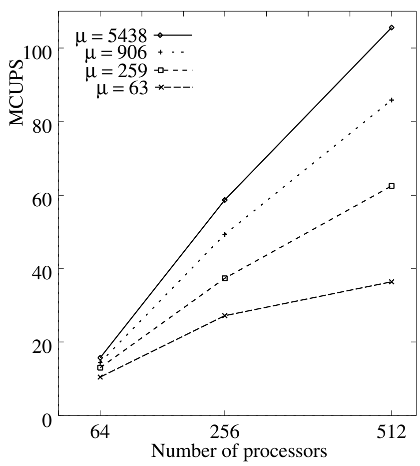

Figure 7.14: Training speed for NETTALK network plotted as a function of the number of processors.

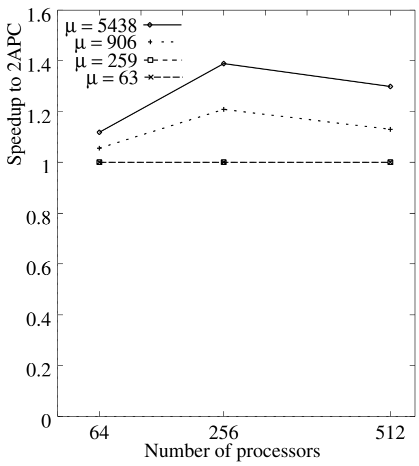

Figure 7.15: Speedup of the best configuration compared to the implementation with the largest N_{NP} value, i.e. the (C_x, C_y) configuration.

mapping.

7.4.1.2 Estimation of the Performance

The accuracy of the estimation model presented in Section 7.3.2.2 is investigated in this section. The training performance of the flexible mapping is tested using the NETTALK application with 120 hidden units.

The results will state if the estimation method is accurate enough to predict the execution time – compared to using real measurements, for selecting the best mapping for a given neural network application. If the estimation method is accurate, it can also be used to predict the possible performance for larger non-existing configurations of the parallel machine.

To measure the average time of various operations, several tests were undertaken in the implemented programs. The average timing used for the following estimations are given in Table 7.3.

Table 7.3: **Time measured for operations in implemented programs on AP1000 (ns).**

t_{add}	1280	Time to add floats sequentially.
t_{mul}	1400	Time to multiply floats sequentially.
$t_{mul+add}$	1770	Time for alternating multiply and add of floats.
$t_{sigmoid}$	7500	Time to compute $1/(1+e^{-x})$.
$t_{comm}(B)$	$60000 + 250B$	Communication time for sending B bytes between cells.

Early experiments revealed underestimated weight update time, and therefore, the constant k (in Expression 7.14) was introduced and set equal to 1.5 in the following experiments. A cache hit prediction has not been implemented in the model as the cache performance analysis on AP1000 showed that the average hit-rate was about 99% $\pm$ 0.5%.

Figure 7.16 and Figure 7.17 show the error profile for the execution time estimates for 64 PEs and 512 PEs, respectively. The estimates are seen to be quite accurate, with an average error less than 5%. For 512 cells the estimated time for the small weight update interval $\mu = 63$ is greater than that measured. For μ= 906 and μ= 5438, on the other hand, the estimates are less than the measured values. On 64 cells, μ equal to 64 is also underestimated for some

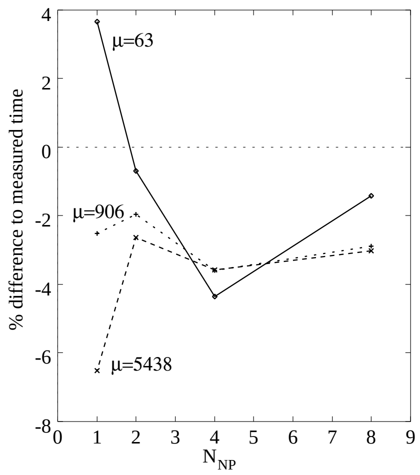

Figure 7.16: Deviation of estimate against measured time for the 64 cell configuration for different weight update intervals.

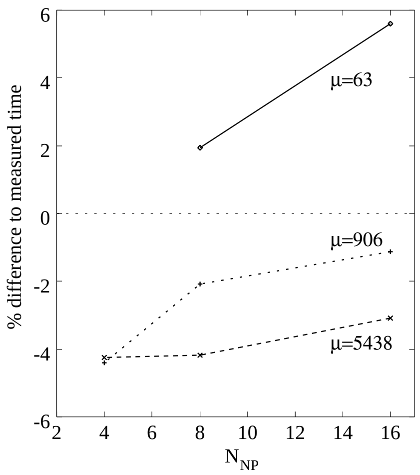

Figure 7.17: Deviation of estimate against measured time for the 512 cell configuration for different weight update intervals.

configurations. This is due to more training patterns – compared on 512 cells, computed between weight updates. Hence, the impact of weight update is less for the small system. Therefore, to improve the model, the training time without weight updating must be increased and the weight update time decreased.

The time estimates can be used to calculate the performance in MCUPS. Figure 7.18 compares the estimated and measured performance for the fastest configurations for 64, 256, and 512 PEs and $\mu = 906$. The N_{NP} values for the fastest mappings are listed in Table 7.4. The estimated performance is slightly higher than the measured performance. Considering the simplicity of the estimation model i.e. only based on floating point operation count and a simple expression for communication time, the estimation method is fairly good. With

Number of processors	N_{NP}
64	4
256	8
512	8

Table 7.4: **The number of N_{NP} values used in Figure 7.18.**

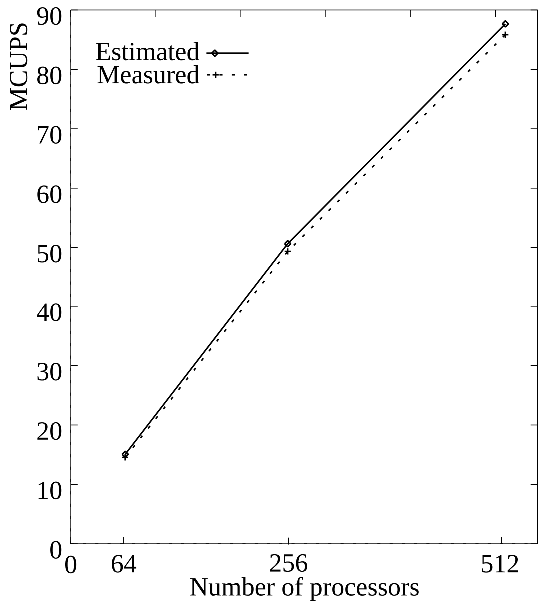

Figure 7.18: **Comparing the fastest configuration for 64, 256 and 512 processing elements to estimated values for a weight update interval equal to 906.**

this level of accuracy, it is interesting to estimate the performance for larger systems.

Figure 7.19 shows the estimated training performance for systems larger than 512 PEs. The configurations with the highest performance are used in the figure. The weight update interval highly influences the obtainable performance. The real difference in performance for the given weight update intervals is probably smaller, according to the estimation error in Figure 7.17. For $\mu = 63$ the MCUPS value is probably underestimated, while for the other it is probably overestimated.

The performance for even larger systems are shown in Figure 7.20. *Learning by epoch* gives a maximum training speed of 3.6 GCUPS, while the weight update interval of $\mu = 906$ shows a maximum of 1.3 GCUPS. Both are for 500K cells. Thus, the performance could be at least ten times faster than that for 512 cells if there were no limitations on the number of cells. This shows the properties of the mapping on a massively parallel computer.

The estimations do not take into account the additional overhead incurred when *few* connections and training vectors are computed by a cell. Thus, the estimates are probably slightly too high for large systems.

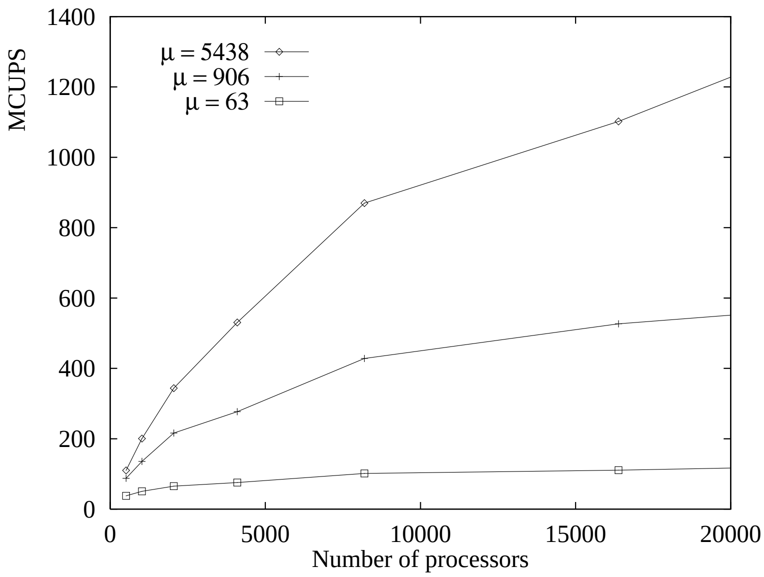

Figure 7.19: **Estimated training performance for the NETTALK application for three different weight update intervals, μ.**

Figure 7.21 presents a performance comparison of the fixed (C_x, C_y)

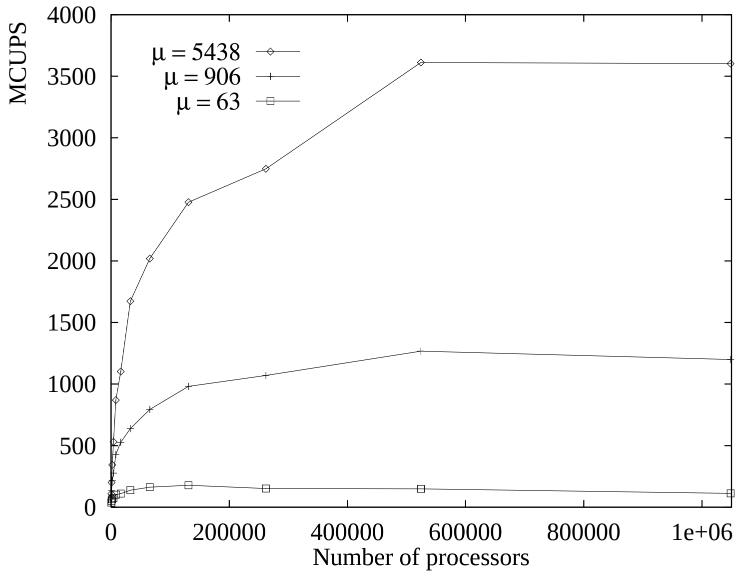

Figure 7.20: **Estimated training performance for NETTALK application for three different weight update intervals, μ.**

configuration – i.e. no training set parallelism in the vertical dimension, with the fastest possible training speed. As the number of PEs increases the gap between the two increases. This emphasizes the importance of *application adaptable* mapping on large parallel systems.

Using the technical data for AP1000 given in Section 7.3.2.2 we can estimate the maximum theoretical training speed of the available systems – as shown in Figure 7.22. For 64 cells the measured performance is 36.4% of the maximum performance, where $\mu = 906$. The difference in performance on 512 cells is 38.5%. Therefore, the implementation could, in the best case, be optimized by a factor of slightly less than three. However, it is anticipated this would require some assembly programming in the code.

This simulation has been tuned to the NETTALK application. A

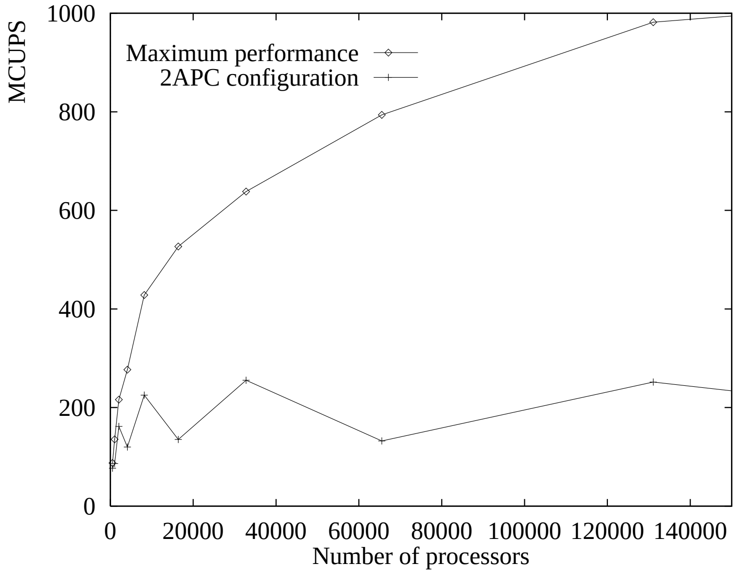

Figure 7.21: **Comparison of the maximum performance of the flexible configuration to the (C_x, C_y) configuration, for $\mu = 906$.**

few experiments were undertaken with an image compression and a speech recognition networks. To obtain accurate estimates, k had to be set to a specific value for each application. Thus for making a general estimation method, an expression for computing k ought to be developed. This should be based on the network size, since its purpose is to compensate for added overhead (communication conflicts) when large weight change matrices are to be added. In conclusion, the experiments are promising for using this estimation technique in picking the best parallel configuration for a neural network application.

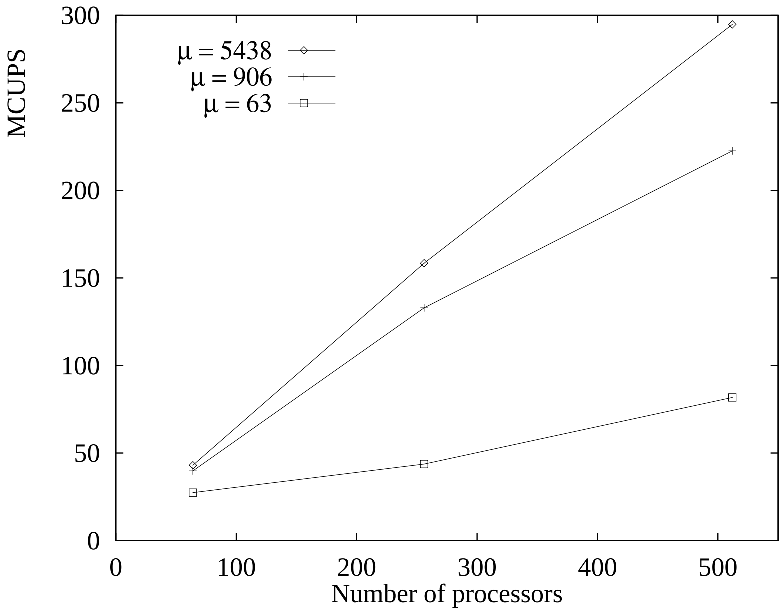

Figure 7.22: **Maximum possible training speed on AP1000.**

7.4.1.3 The Convergence of NETTALK Learning

To investigate the convergence rate for different weight update intervals, 1000 of the most common English words (a total of 5438 characters) were used. The error function for a training pattern p is equal to Equation 1.4 and is given by

$$E_p(n) = \frac{1}{2}\sum_k (d_{p,k} - y_{p,o,k})^2$$

where n is the training iteration number, $d_{p,k}$ is the target output and $y_{p,o,k}$ is the output layer output. Rumelhart et al. use the values 0.1 and 0.9 as target values, since the extreme values of 0 or 1 can never be reached [36]. If the error is 0.1 per output unit the pattern error becomes

$$E_p(n) = \frac{1}{2}\sum_{k=1}^{26}(0.1)^2 = 0.13$$

and this value has been used as a threshold for determining if a pattern is trained or not. Weight change values were accumulated for a pattern p having $E_p(n) > 0.13$, while a pattern is trained when $E_p(n) \leq 0.13$. This is a slightly different error measure than reported by Sejnowski and Rosenberg, who measured the error at each individual output neuron. They used a "best guess" which compared the output values to a subset of the training set (52 target vectors). If the vector that had the smallest output error was equal to the training vector then the vector was considered correctly trained. Both methods decrease the sum of square error function

$$E(n) = \sum_{p=1}^{5438}\sum_{k=1}^{26}(d_{p,k} - y_{p,o,k})^2 \tag{7.16}$$

The error measure used in this research avoids over-training without a large loss in the load balance of the computation. Since the purpose of these experiments has been to study the relation between weight update interval and convergence and not the performance of the application, a separate test set has not been used. Testing convergence on the training set requires less time than using a test set. Thus, more tests for each weight update interval could in this way be undertaken.

For each weight update interval investigated, the best learning rate η was searched for. Between 10 and 15 runs of each weight update interval were performed. These experiments in total required several hundred CPU hours. For a small μ, the best learning rate was equal to 1.5. It had to be decreased for larger values of μ. For larger μ, the convergence was more sensitive to the selection of η. The best value for the smoothing term α was, after some initial experiments, found to be 0.9. This value was used for all the experiments. Due to this smoothing term, it was possible to keep a fairly large learning rate value even with infrequent weight updates. Bias was also used in the experiments.

Experiments with adaptive learning rate during training and other similar methods to speed up the convergence are not included. This is mainly because it would have required a *much* larger number of experiments to make a fair comparison of the convergence for different weight update intervals.

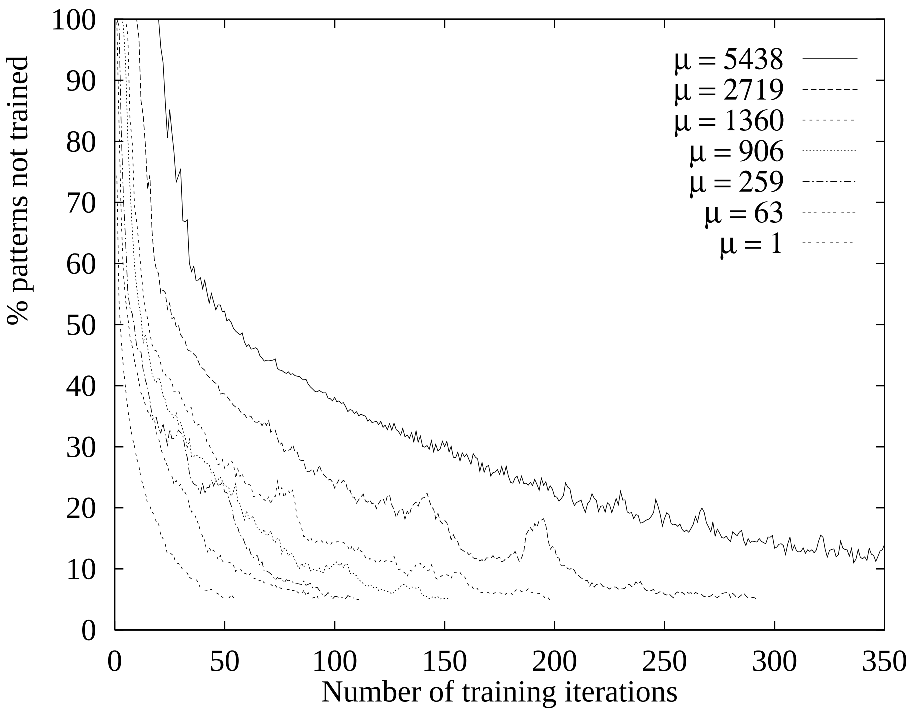

Figure 7.23: **Percentage of characters that are not trained for various weight update intervals.**

Figure 7.23 shows the results of the convergence as a percentage (called $E_{\%}$) of the characters that have not been trained. Each curve represents one weight update interval. Almost without exception, a larger μ implies a slower reduction in number of non-trained patterns. The percentage of characters not trained went down and stabilized slightly below 5%. This is due to the representation limitation of the network as mentioned in [3]. Therefore, a network is here regarded as converged, when $E_{\%} < 5\%$.

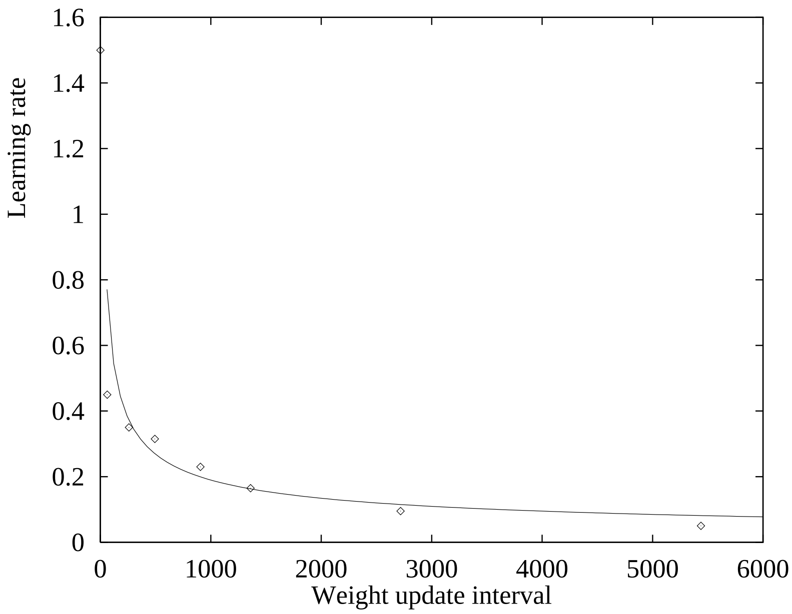

Figure 7.24: **The best learning rate (η) for each investigated weight update interval. The curve plots the derived rule.**

Figure 7.24 shows the optimal learning rate – the value resulting in the smallest number of iterations for convergence, for each of the μ used in the experiments. Based on these values the following rule is established

$$\eta(\mu) = 6.0\mu^{-0.5} \tag{7.17}$$

This is comparable to the rule derived by Paugam-Moisy in [26]: $\eta(\mu) = k_1\mu^\alpha$ for the application of classifying patterns into three classes – see Section 1.3.1.3.

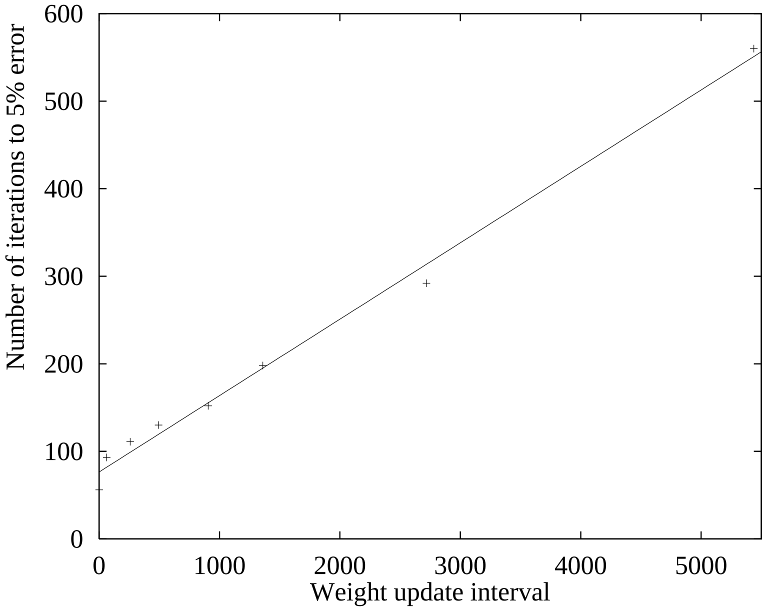

Figure 7.25: **Number of iterations needed to obtain convergence with $E_{\%} < 5\%$ as the stopping criteria.**

The number of iterations required for convergence is given in Figure 7.25. Due to the fairly linear distribution of the points, the least squares method is used to establish the experimental rule

$$N(\mu) = \frac{\mu}{11.5} + 76.4 \tag{7.18}$$

This linear relation between number of training iterations and the weight update interval was also found by Paugam-Moisy. The linear distribution of points tells us that for predicting $N(\mu)$ it is sufficient to estimate the total number of iterations for a few (minimum 2) weight update intervals. These can then be used to find a linear formula for $N(\mu)$.

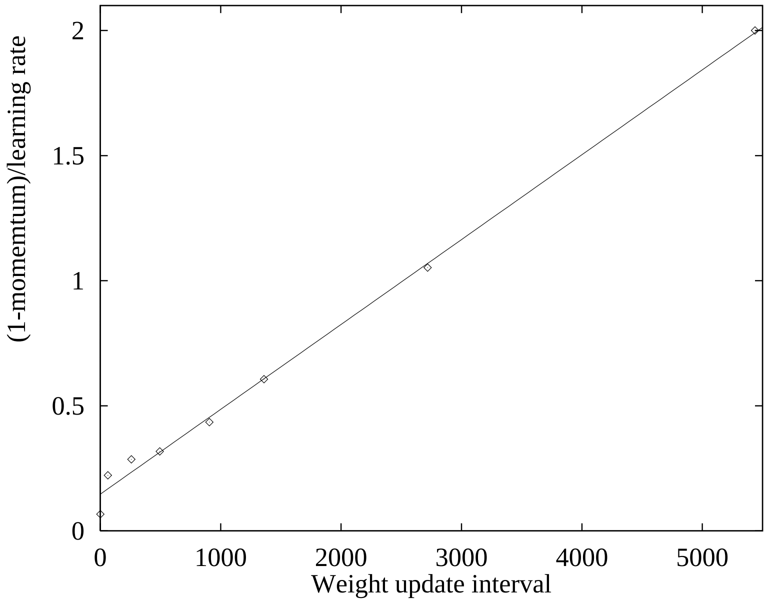

Figure 7.26: **Plotting of** $(1-\alpha)/\eta$**.**

As mentioned in Section 7.3.2.1 several researchers have found that N is proportional to $(1-\alpha)/\eta$. Figure 7.26 plots this expression for the learning rates in Figure 7.24. The points are linearly distributed along the least squares fit line. By scaling by the number of iteration for $\mu = 906$ we get

$$N(\mu) = \frac{\mu}{8.43} + 51.2 \tag{7.19}$$

This gives a good estimation for $\mu = 1$ with a number of 5 iterations less than measured. However, the estimates are less precise for larger weight update intervals, e.g. for $\mu = 5438$ a number of 85 iterations more than measured is estimated.

In the following it is shown that the total number of iterations needed can be estimated based on the error after a small number of training iterations. By looking at the curves in Figure 7.23, we see that they are of similar form, but have different curvature. To find a curve that estimates each error curve, logarithmic regression can be used to find an expression for each μ_i of the form

$$E_{\%}(n) = k_1 n^{k_2} \tag{7.20}$$

where n is the iteration index and k_1 and k_2 are parameters to be estimated. This requires that the points $(\log n_i, \log E_{\%}(n_i))$ $\forall i$ are situated on a straight line. Figure 7.27 shows a logarithmic plot of Figure 7.23.

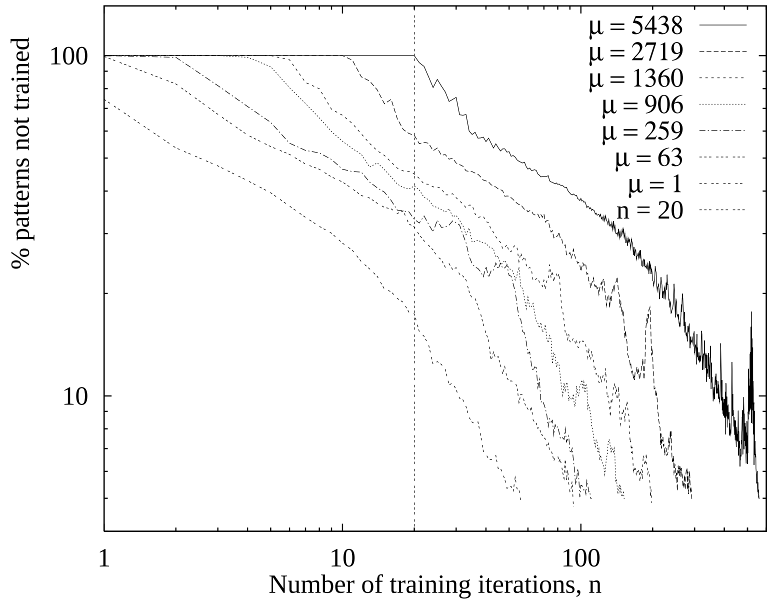

Figure 7.27: **Convergence of NETTALK, logarithmic plot. A vertical line is plotted for $n = 20$.**

Above 20 iterations – see on the right-hand side of the vertical line in the figure, the curves are becoming more linear for increasing iterations. Thus, regression should be used on this linear part.

Several different regression analyzes were made to search for a curve describing the error convergence. Equation 7.20 was extended

$$E_{\%}(n) = k_1(n + k_3)^{k_2} \tag{7.21}$$

by k_3, which may be called the offset of the error curve, equal to twice[4] the number of iterations from start of training until $E_{\%} \leq 95\%$. Equation 7.21 was solved for each μ for the initial point ($n = 25, E_{\%}(25)$) and the convergence point (n when $E_{\%}$ becomes less than 5%, 5) with respect to k_1 and k_2. The points were given by the registered values during training – see Table 7.5.

Table 7.5: **The data on NETTALK convergence used for estimation.**

Weight update interval (μ)	k_3	$E_{\%}(25)$	n when $E_{\%}(n) < 5$
1	0	12.61	56
63	0	25.65	93
259	4	32.86	111
494	6	32.38	130
906	8	35.77	152
1360	12	41.12	198
2719	22	53.51	292
5438	42	85.20	560

The value of k_2 was found to be fairly constant for all the weight update intervals and averaged to approximately -1.25. Now k_1 can be computed based on *only* the error at $n = 25$ iterations

$$k_1 = \frac{E_{\%}(n)}{(n + k_3)^{-1.25}} \tag{7.22}$$

[4] Experiments showed a better approximation when using twice the number, compared to using the explicit number.

Furthermore, the total number of iterations is given from Equation 7.21 by

$$N = \left(\frac{5}{k_1}\right)^{1/-1.25} - k_3 \tag{7.23}$$

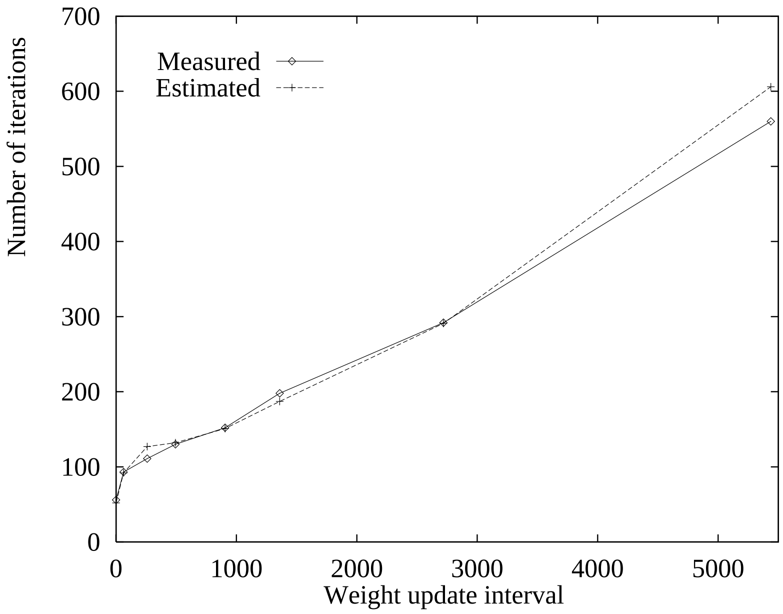

Figure 7.28: **Comparing measured and estimated values of $N(\mu)$.**

Figure 7.28, which shows both measured and estimated values of $N(\mu)$, proves the accuracy of the estimation model for this application. However in general this accuracy may not be the case. This is a field of research where very little published work using real applications exists. Thus, more studies are required to form a more general framework and to see if it is possible to estimate the convergence in general.

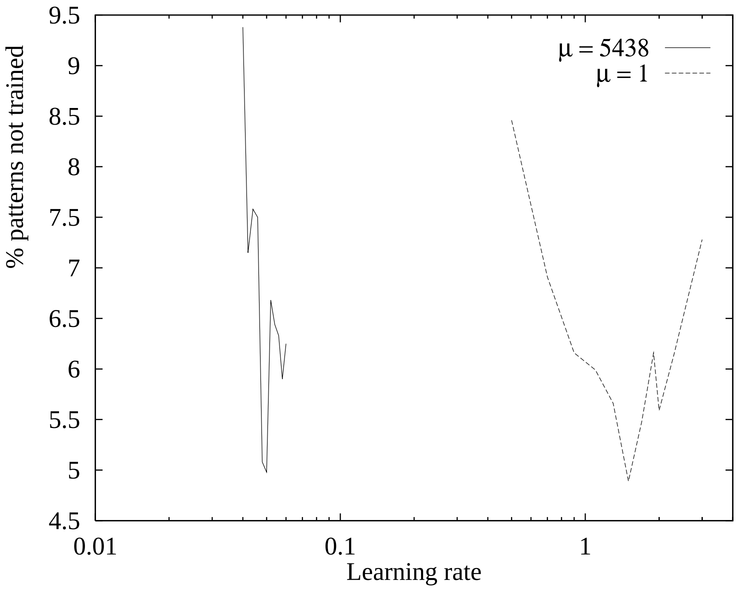

Figure 7.29: **The sensitivity of the learning rate (η) on the convergence for epoch versus pattern learning.**

Sensitivity and Convergence Aspects The BP algorithm is more sensitive to the selection of the learning rate when the weights are infrequently updated. Figure 7.29 shows the convergence for $\mu = 1$ (*learning by pattern*) and $\mu = 5438$ (*learning by epoch*). The error $E_{\%}$ is plotted and the minimum error corresponds to the learning rate which gives the quickest convergence – i.e. $E_{\%}$ is less than 5%. The x-axis is logarithmic, and still the epoch learning has a narrower valley in the error curve.

In the training scheme used, a pattern is said to be trained if the error is below a certain threshold. Since the weights are changed during an iteration, there is a risk of trained patterns becoming de-trained, i.e. have an error larger than the threshold after the training iteration. Thus, the number of patterns trained correctly may be less

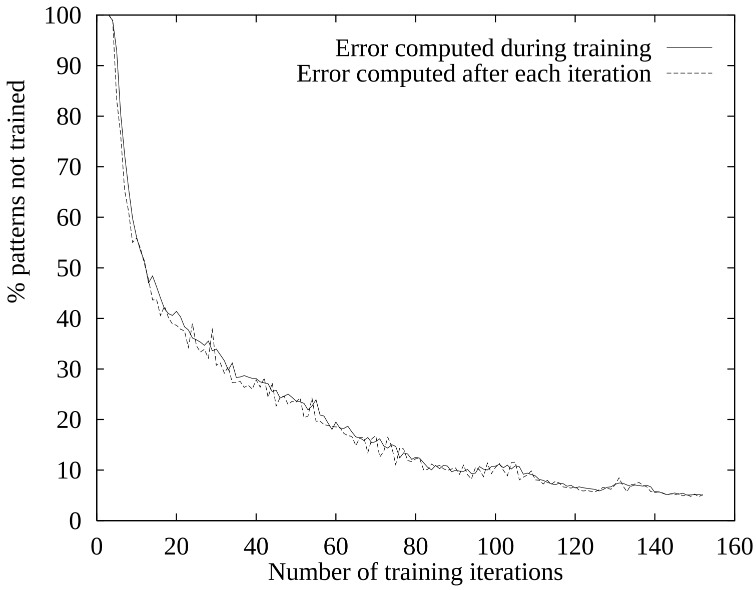

Figure 7.30: **Error computed during training compared to error computed after each iteration (epoch) for $\mu = 906$.**

than the number counted during an iteration. Figure 7.30 compares the $E_{\%}$ accumulated *during* each training iteration against $E_{\%}$ computed for the whole training set *after* each training iteration for $\mu = 906$. Even though the epoch computed error is less stable, the two convergence criteria harmonize well with little difference in the final convergence. Similar results were recorded for other values of μ.

Figure 7.31 shows the convergence when $E_{\%} < 10\%$. 31 iterations are required before convergence using *learning by pattern*. In the NETTALK paper [3], the training continued for 30 iterations. As for the $E_{\%} < 5\%$ convergence criteria, there is a linear relation between the weight update interval and the number of iterations required before convergence.

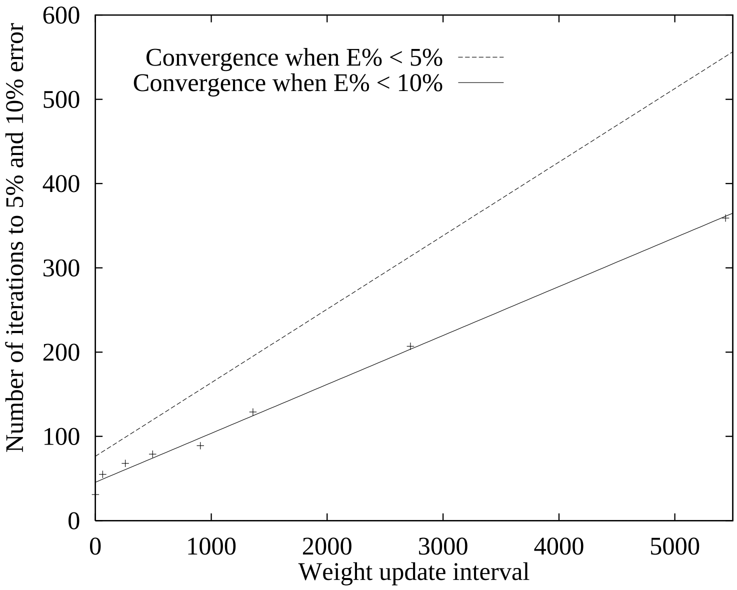

Figure 7.31: **Comparing $E_{\%} < 5\%$ and $E_{\%} < 10\%$ error criteria.**

An interesting aspect is the ratio

$$k_e = \frac{N(P)}{N(1)}, \tag{7.24}$$

where P is the number of patterns in the training set. Then, $N(P)$ represents the number of iterations for *learning by epoch*. For NETTALK training with 5% error threshold

$$k_e = \frac{560}{56} = 10 \tag{7.25}$$

For a 10% error threshold this becomes

$$k_e = \frac{359}{31} = 11.58 \tag{7.26}$$

For the least strict threshold (10%), there is a larger difference in number of iterations for *learning by pattern* compared to *learning by epoch*.

7.4.1.4 Minimizing the Total Training Time

In this section, the best weight update intervals for running NETTALK on AP1000 is determined. By applying Equation 7.5, the total training time is found as the time for one iteration multiplied by the number of iterations used for convergence. Total time calculations are conducted for μ= 63, 259, 494, 906, 1360, 2719, and 5438.

The time for one iteration, T_{1it}, can be computed from Figures 7.11 – 7.13 (not all μ-values are shown). The number of iterations needed for convergence, $N(\mu)$, is given in Figure 7.25.

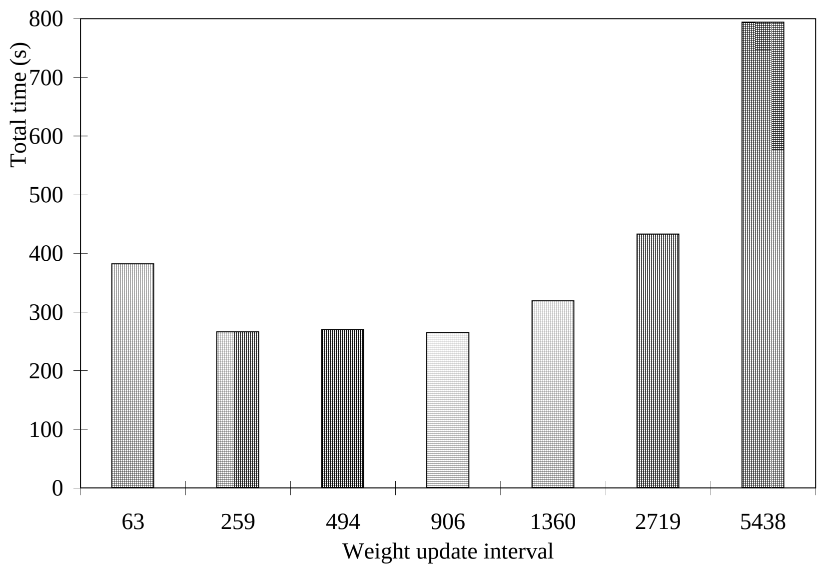

Figure 7.32: **Total training time for NETTALK running on 512 cells, using 120 hidden units.**

For the 512 cell system, the total training time is plotted in Figure 7.32. The best weight update interval is for $\mu = 906$ using the configuration $(N_{TSP}, N_{NP}) = (64, 8)$. Convergence is obtained after

265 seconds. However, there is only a marginal differences in the time for $\mu \in \{259, 494, 906\}$. The slowest training time is for $\mu = 5438$. Thus, infrequent weight update is not as promising as the MCUPS measure indicates.

Weight changes have not been accumulated for a pattern where the error is sufficiently small, to avoid overtraining. Thus, T_{1it} decreases during training as the number of patterns trained increases. As a constant T_{1it} is used, the T_{total} is larger than the real execution time.

Running the same program – with parallel parts removed, on a Sparc10 workstation using *learning by pattern*, reached convergence after 56 iterations and 115 minutes. This was measured when no other users were active on the computer. AP1000 speedup based on total execution time ($T_{1it}N$) is therefore shown to be 26 times. This is a conservative measure, since the decreasing T_{1it} for AP1000 is not accounted for. When the serial program updated the weights for every pattern, the total training time was 236 minutes. This is 54 times slower than training on AP1000. Thus, total training time can be reduced from hours to minutes by using parallel processing.

Total training time for 256 cells and 64 cells are given in Figures 7.33 and 7.34, respectively. As the system gets smaller, more frequent weight updates lead to the shortest training time. This can partly be understood by the number of training set partitions, as explained in Section 7.4.1.1. In Figure 7.11 the difference in maximum performance for the various μ values is small compared to Figure 7.13. Combined with the few iterations required for convergence for frequent weight updating, this leads to the shortest total training time for small systems.

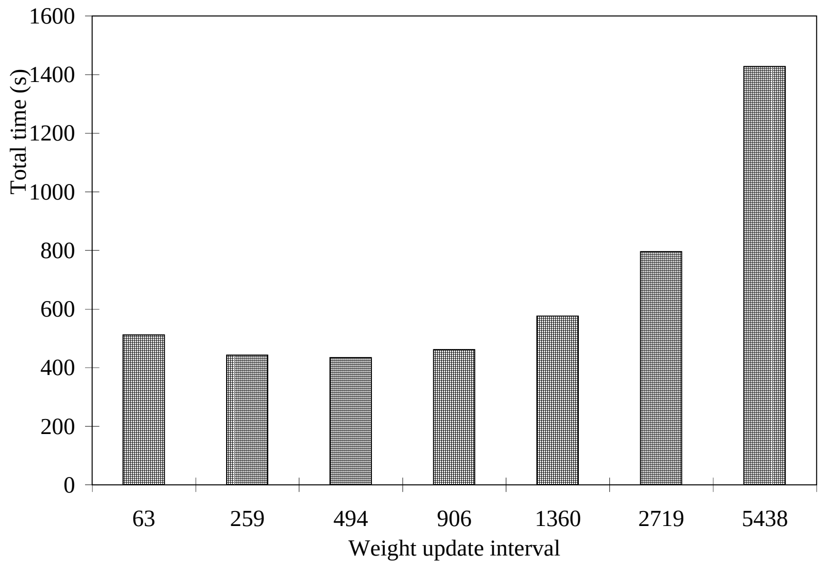

Figure 7.33: **Total training time for NETTALK running on 256 cells, using 120 hidden units.**

7.4.2 Sonar Target Classification

In this section, the results on the convergence of the sonar return application, described in Section 2.2.3.3, are given. In the experiments, all the available patterns (208) have been used as a training set. The weights are updated – or accumulated for *learning by block*, for a pattern where the error is larger than 0.04. That is

$$E_p(n) > \frac{1}{2}\sum_{k=1}^{2}(0.2)^2 = 0.04$$

This threshold is based on an error of 0.2 on each of the output units, which was used by Gorman et al. [6, 7]. As in their work, the smoothing term was set to 0. The number of hidden units was set to 24, which was reported as giving the best recognition rate on the

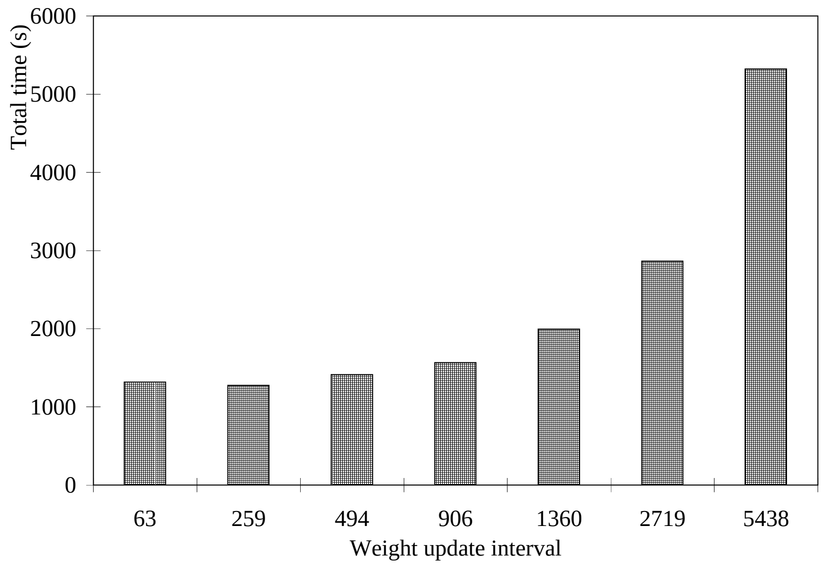

Figure 7.34: **Total training time for NETTALK running on 64 cells, using 120 hidden units.**

training set.

7.4.2.1 Results on Sonar Target Classification

The convergence for different weight update intervals is given in Figure 7.35. The y-axis represents the percentage of training patterns with $E_p > 0.04$. The results are very similar to NETTALK – a larger weight update interval leads to slower learning. However, the convergence curves are steeper and it is not possible to use the same logarithmic regression as for NETTALK to estimate the number of training iterations.

The convergence was very sensitive to small changes in initial

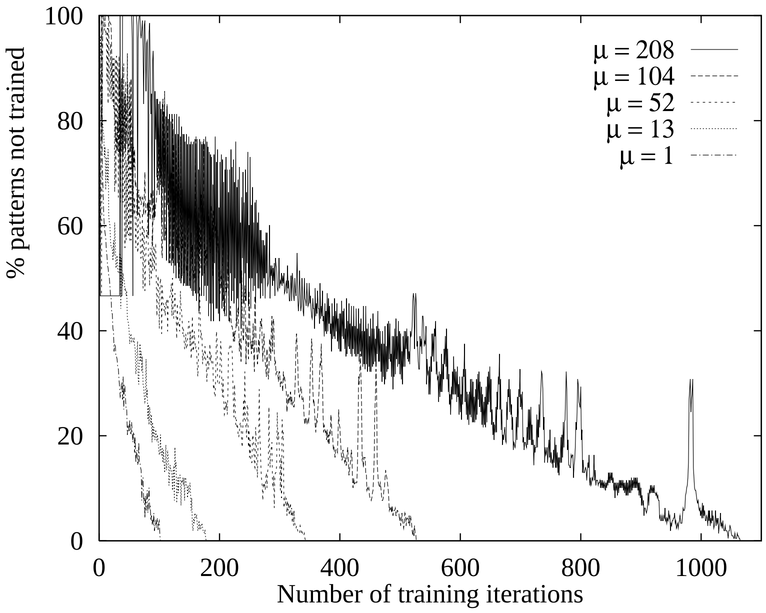

Figure 7.35: **The percentage of patterns that have $E_p > 0.1$, for learning sonar return classification.**

weights and other parameters. It was seen that the resolution of floating point variables influenced the convergence rate. Continuous values in the input units, instead of two-level values as in the case of NETTALK, probably make the algorithm more sensitive to the accuracy of the variables.

The best learning rate for each weight update interval is plotted in Figure 7.36. A similar experimental rule to Equation 7.17 is established

$$\eta(\mu) = 3.0\mu^{-0.5} \tag{7.27}$$

The total number of iterations until *all* training patterns have reached $E_p \leq 0.04$ during a learning cycle is plotted in Figure 7.37

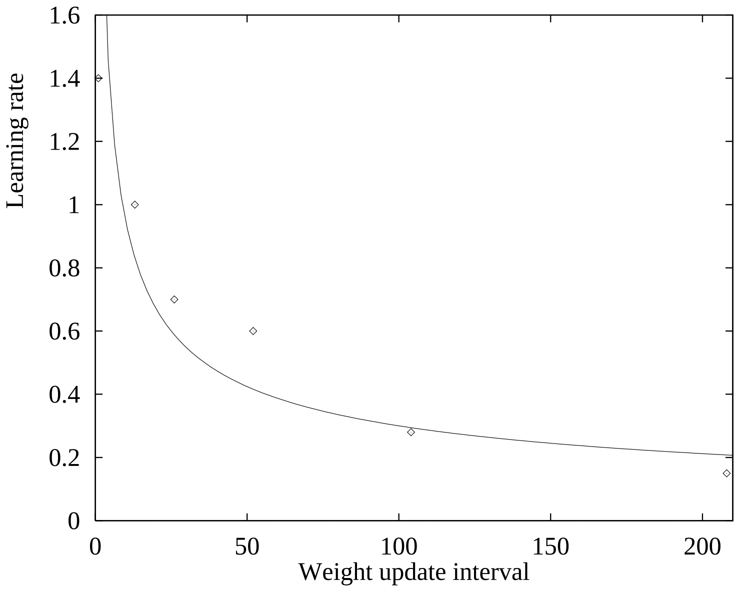

Figure 7.36: **The best learning rate η for each of the investigated weight update intervals. The curve plots the derived rule.**

together with the derived least squares fit of the points

$$N(\mu) = 106.66 + 4.51\mu \tag{7.28}$$

The number of iterations required for an error threshold equal to 5% not trained patterns is plotted in Figure 7.38. This is equivalent to the error threshold used for NETTALK. The least squares error equation is given by

$$N(\mu) = 97.04 + 3.88\mu \tag{7.29}$$

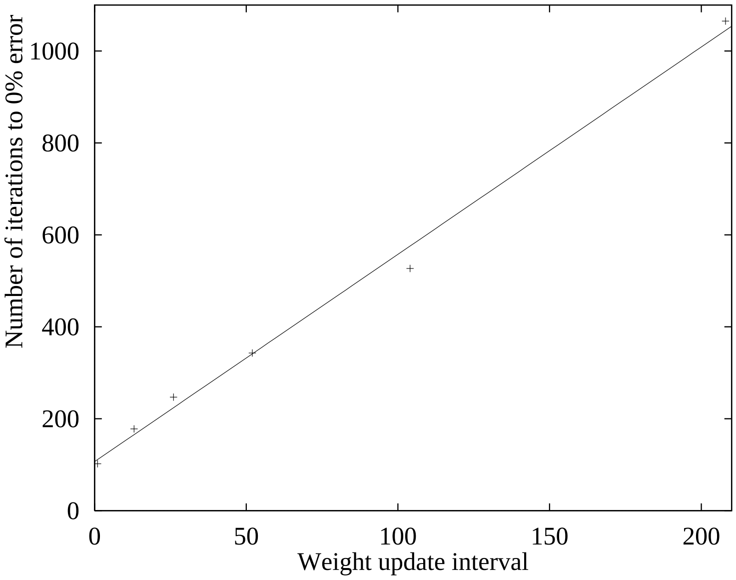

Figure 7.37: **Number of iterations required to obtain convergence, i.e. $E_p \leq 0.04$ for all training patterns.**

7.4.2.2 Comparing Sonar Results to NETTALK Results

In this section, common properties of the two applications are highlighted. The idea is to search for a general way of estimating the number of iterations needed for convergence, N, as a function of the weight update interval.

For NETTALK, the learning ratio was found to be $k_e = 10$ for a 5% error threshold and $k_e = 11.58$ for a 10% error threshold. Correspondingly for the sonar target classification (all of the patterns trained) we have

$$k_e = \frac{1065}{102} = 10.44 \tag{7.30}$$

For the less strict stopping criteria of 5%, $k_e = 11.9$. Thus, k_e is

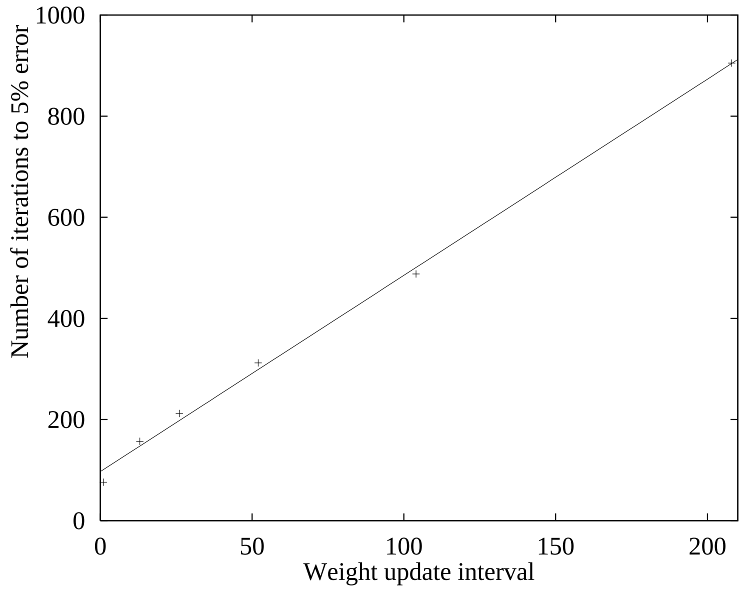

Figure 7.38: **Number of iterations needed to reach a stopping criteria of 5% error.**

in the 10-12 range for both applications. Thus, *learning by pattern* trains 10-12 times faster than *learning by epoch*. Moreover, a less strict error threshold increases k_e. The resemblance between these two applications was not expected, since they are different in both network size and number of training patterns.

This result, of a near constant k_e ratio, will now be used in estimating $N(\mu)$ based on a single result for convergence, $N(1)$. The relation between μ and the k-ratio is illustrated in Figure 7.39. The straight line is assumed according to the earlier shown linear relation between the weight update interval, μ and $N(\mu)$. The equation for the line passing through the points (1,1) – *learning by pattern* and

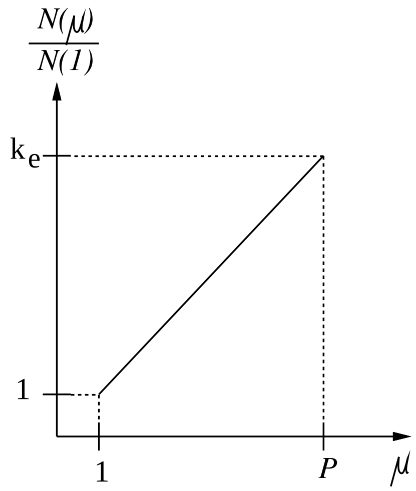

Figure 7.39: **The relation between μ and k_e, where P is the total number of patterns in the training set.**

(P, k_e) – *learning by epoch* is given by

$$k_\mu - 1 = \frac{k_e - 1}{P - 1}(\mu - P) \Leftrightarrow k_\mu = \frac{k_e - 1}{P - 1}(\mu - P) + 1 \qquad (7.31)$$

where k_μ also can be understood as $N(\mu)/N(1)$. An estimate of the $N(\mu)$ is then given by

$$N(\mu) = k_\mu N(1) \qquad (7.32)$$

For neural training, the application is usually trained several times with different parameters. Therefore, Equation 7.32 can be used for the approximation of $N(\mu)$, after some initial convergence tests, in the heuristic shown in Figure 7.10. To confirm the estimation method, other applications should be tested.

To set the appropriate learning rate, the previous experimental rules (Equation 7.17 and 7.29) can be formulated in a general approximation rule

$$\eta(\mu) = k\mu^{-0.5} \tag{7.33}$$

where k is a positive constant less than 10. As the training set gets smaller, the appropriate k value decreases. This is in accordance to the two experiments reported in this chapter.

7.4.3 Speech Recognition Network

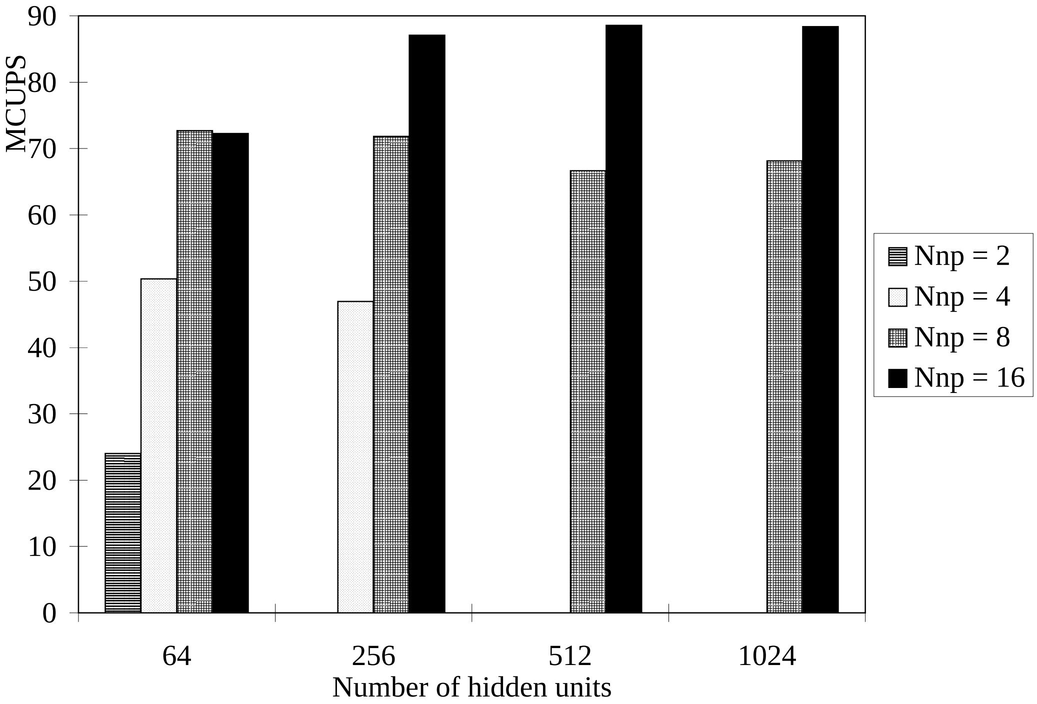

Figure 7.40: **Training performance on 512 processors for speech recognition network. N_{NP} = 2, 4, 8, and 16.**

Figure 7.40 shows the performance for a speech recognition network on 512 processing elements. It was not possible to execute all the configurations due to communication limitations (*ring buffer overflow*). Except for 64 hidden units, the highest training speed is obtained for $N_{NP} = 16$ i.e. 16 cells assigned to each copy of the network. For this configuration there is no training set partition in the horizontal dimension and therefore, the training speed does not exceed that for the fixed (C_x, C_y) configuration. To further improve the performance, synapse parallelism should probably be introduced due to the size of the network.

7.4.4 Image Compression

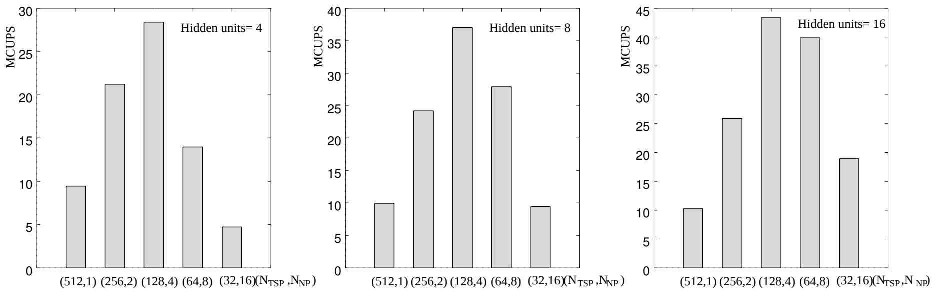

Figure 7.41: **MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 32 processor configuration training a compression network.**

Image compression requires a small network [4], but implies a large amount of data, both for training and for real-time compression. The performance for a 64 input and 64 output network and 512 cells is given in Figure 7.41 for three different compression ratios. The number of hidden units varied between 4, 8, and 16. A small number of hidden units implies a large compression ratio. The weights were updated for every 512 training patterns. For the smallest network (4 hidden units) the performance of the non-contention mapping

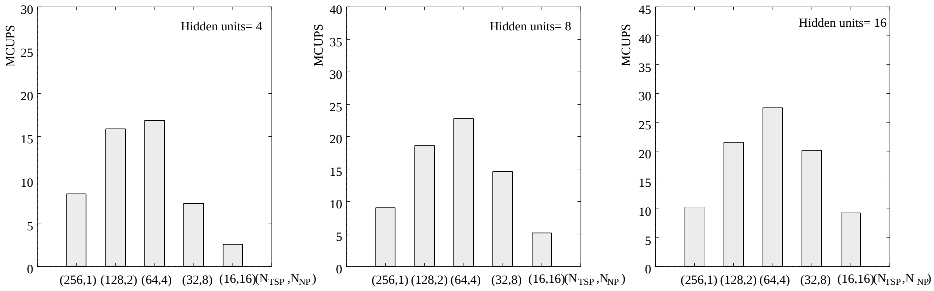

Figure 7.42: **MCUPS performance for different combinations of neuron and training set parallelism on a 16 x 16 processor configuration training a compression network.**

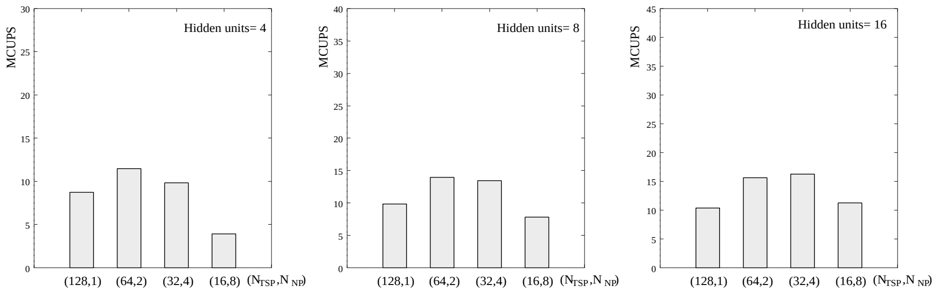

Figure 7.43: **MCUPS performance for different combinations of neuron and training set parallelism on an 8 x 16 processor configuration training a compression network.**

$(N_{TSP}, N_{NP}) = (32, 16)$ is approximately 6 times less than that for the best mapping $(N_{TSP}, N_{NP}) = (128, 4)$. Thus, for small networks a large degree of training set parallelism is beneficial.

Figure 7.42 shows the performance for 256 cells. In this configuration, $N_{NP} = 4$ also gives the best overall performance. However, for $N_{NP} = 2$ the performance is higher than for $N_{NP} = 8$ – for 8 and 16 hidden units, which is opposite from Figure 7.41.

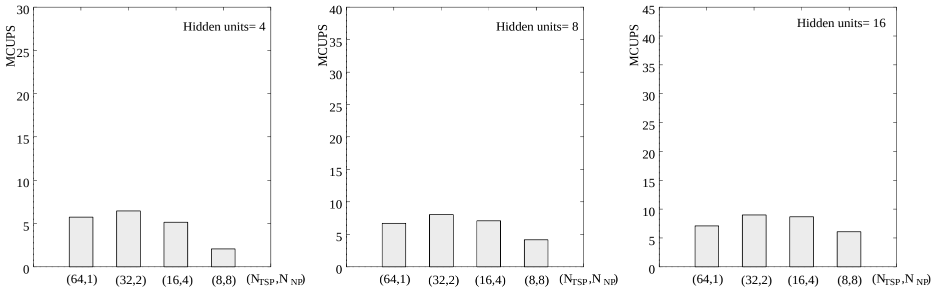

Figure 7.44: **MCUPS performance for different combinations of neuron and training set parallelism on an 8 x 8 processor configuration training a compression network.**

The performance on 128 processors appears in Figure 7.43. The number of cells in the y-dimension (16) and x-dimension (8) is half of that in the 512 cell system. The difference between the best and worst results is lower than for the larger systems. For 4 and 8 hidden units the maximum learning speed is reached when $N_{NP} = 2$. Thus, assigning fewer processors to each copy of the network is important as the number of cells is reduced. In this way, a large number of training set partitions can be maintained. The training speed on 64 cells is given in Figure 7.44.

The highest speeds in each of the above experiments are plotted in Figure 7.45. The ratio between the highest performance achieved and the performance of the (C_x, C_y) configuration is plotted in Figure 7.46. As shown, a reduction in the number of hidden units leads to larger ratios. For 4 hidden units and 256 processors the difference is a factor of 6.5. This corresponds to 550% faster training speed – a considerable difference. The nonlinearity in the curves in the figure can be explained by the difference in configuration. For 128 and 512 processors $C_y = 2C_x$, while for 64 and 256 processors $C_y = C_x$. Thus,

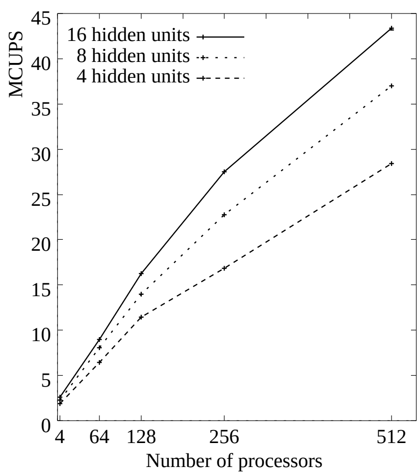

Figure 7.45: Training speed for image compression network plotted as function of the number of processors.

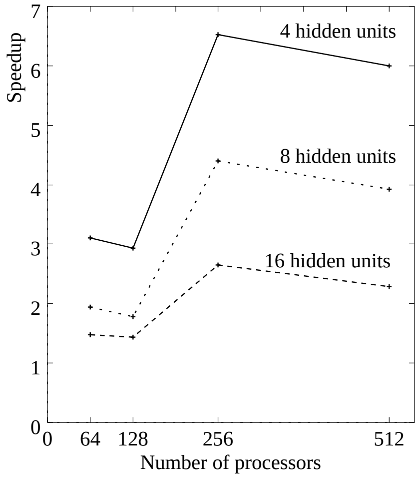

Figure 7.46: Speedup of the best configuration compared to the (C_x, C_y) configuration.

when there are twice the number of processors in the vertical dimension, additional training set partitions in the horizontal dimension do not give rise to the same increase the performance as for 64 and 256 processors. It can be seen that the scaling in Figure 7.45 would drop dramatically if the (C_x, C_y) configuration had been used. This is especially true for networks with a small number of hidden units, for which the flexible mapping already has a limited scaling.

The primary goal of designing the parallel mapping scheme has been to reduce the neural network training time. However, for image compression there will be a large amount of data to be processed for real-time compression of video frames. The proposed mapping will allow fast compression by compressing different parts of each frame on different copies of the neural network.

7.4.5 Conclusions on the Application Adaptable Mapping

In this chapter, a near optimal mapping of the back propagation algorithm onto a parallel computer has been proposed. Multiple degrees of parallelism have been exploited and the influence of the weight update interval on convergence has been included in the mapping strategy.

The importance of the flexible implementation depends on the neural application. For NETTALK, 40% higher performance than for the (C_x, C_y) configuration was achieved in the best case. For the smaller networks like image compression network the difference in performance was larger – 550% in the best case. This highlights the advantages of an *application adaptable* mapping algorithm to reduce training time.

The results also indicate that the flexibility becomes more important for parallel computers with a large number of processing elements. This is because most neural network applications require a relatively small neural network size. However, the number of training patterns for training or the amount of data to be processed in the recall-phase may be very large. Therefore on large systems, the number of training set partitions should be large rather than the number of processors assigned to each subset.

Estimation of execution time, based on the number of floating point operations and communication time estimates, was tested. For the NETTALK application it turned out to be accurate and was used to show that the number of processors for the flexible mapping can be increased almost without limit resulting in increased performance.

Most other BP implementations published do not consider the weight update interval or the mapping strategy according to the *given* neural network application. The convergence tests of two applications in this chapter indicate a linear relation between the weight update interval and the number of iterations to reach convergence. *Learning by epoch* is a factor of 10-12 times slower than *learning by pattern*. Even though the speedups indicated by the MCUPS figures are not practically obtainable, the computation of total execution time undertaken shows that parallel processing can drastically reduce training time. This requires that the parallel implementation is flexible and adaptable to the given neural network and training set.

Bibliography

[1] Hiroaki Ishihata and others, "Third generation message passing computer AP1000," *Proc. of the International Symposium on Supercomputing*, pp. 46–55, Nov. 1991.

[2] M. Kuga and Y. Namiuchi and B.O. Apduhan and T. Sueyoshi, "Implementation and performance evaluation of a neural network simulator on highly parallel computer AP1000," *ICPAS*, 1993.

[3] Terrence J. Sejnowski and Charles R. Rosenberg, "Parallel networks that learn to pronounce English text," *Complex Systems*, vol. 1, pp. 145–168, 1987.

[4] Kwang Bo Cho and others, "Image compression using multilayer perceptron with block classification and SOFM coding," *Proc. of World Congress on Neural Networks*, vol. 3, pp. 26–31, 1994.

[5] Nelson Morgan, "Using a million connections for continous speech recognition," *Proc. of Int. Conference On Neural Network Processing*, pp. 1439–1444, October 1994.

[6] R. Paul Gorman and Terrence J. Sejnowski, "Analysis if hidden units in a layered network trained to classify sonar targets," *Neural networks*, vol. 1, pp. 75–89, 1988.

[7] R. Paul Gorman and Terrence J. Sejnowski, "Learned classification of sonar targets using a massively parallel network," *IEEE Trans. on acoustics, speech and signal processing*, vol. 36, no. 7, pp. 1135–1140, 1988.

[8] J.M. Adamo and D. Anguita, "Object oriented design of a BP neural network simulator and implementation on the Connection Machine (CM-5)," tech. rep., International Computer Science Institute, September 1994. TR-94-46.

[9] G. Blelloch and C. R. Rosenberg, "Network learning on the Connection Machine," *Proc. of IJCAI87*, pp. 323–326, 1987.

[10] Darin Jackson and Dan Hammerstrom, "Distributing back propagation networks over the Intel iPSC/860 hypercube," *Proc. of Int. Joint Conference on Neural Networks*, vol. I, pp. 569–574, 1991.

[11] D. A. Pommerleau and others, "Neural network simulation at Warp speed: How we got 17 million connections per second," *Proc. of IEEE Int. Conference on Neural Networks*, 1988.

[12] E. Sànchez and S. Barro and C.V. Regueiro, "Artificial neural networks implementation on vectorial supercomputers," *Proc. of IEEE Int. Conference on Neural Networks*, (Orlando, FL), pp. 3938–3943, June 28 - July 2, 1994.

[13] Andreas Zell and others, "Problems of massive parallelism in neural network simulation," *Proc. of IEEE Int. Conference on Neural Networks*, pp. 1890–1895, 1993.

[14] Xiru Zhang, "The backpropagation algorithm on grid and hypercube architectures," *Parallel Computing*, vol. 14, pp. 317–327, Summer 1990.

[15] Bernard Faure and Guy Mazare, "A cellular architecture dedicated to neural net emulation," *Microprocessing and Microprogramming*, vol. 30, pp. 249–256, August 1990.

[16] Hideki Yoshizawa and Kazuo Asakawa, "Highly parallel architecture for back-propagation using a ring register data path," *Fujitsu Sci. Tech. J*, pp. 227–233, September 1993.

[17] Jim Torresen and Shin-ichiro Mori and Hiroshi Nakashima and Shinji Tomita and Olav Landsverk, "Parallel back propagation training algorithm for MIMD computer with 2D-torus network," *Proceedings of International Conference On Neural Information Processing (ICONIP'94), Seoul, Korea*, vol. 1, pp. 140–145, October 1994.

[18] S. Y. Kung and J. N. Hwang, "Parallel architectures for artificial neural nets," *Proc. of IEEE Int. Conference on Neural Networks*, vol. 2, (San Diego), pp. 165–172, July 24-27 1988.

[19] Jim Torresen and Hiroshi Nakashima and Shinji Tomita and Olav Landsverk, "General mapping of feed-forward neural networks onto an MIMD computer," *Proc. of IEEE Int. Conference on Neural Networks (ICNN'95)*, (Perth, Western Australia), IEEE, 27 November – 1 December 1995.

[20] Jim Torresen and Shinji Tomita and Olav Landsverk, "The relation of weight update frequency to convergence of BP," *Proc. of World Congress on Neural Networks (WCNN'95)*, vol. 1, (Washington, D.C.), pp. 679–682, INNS Press, July 1995.

[21] Takashi Yukawa and Tsutomu Ishikawa, "Optimal parallel backpropagation schemes for mesh-connected and bus-connected multiprocessors," *Proc. of IEEE Int. Conference on Neural Networks*, pp. 1748–1753, 1993.

[22] Hiroaki Ishihata, "Performance evaluation of the AP1000," *Proc. of CAP workshop*, pp. N–1–8, 1991.

[23] Jim Torresen and Shin-ichiro Mori and Hiroshi Nakashima and Shinji Tomita and Olav Landsverk, "Exploiting multiple degrees of BP parallelism on the highly parallel computer AP1000," *Fourth International Conference on Artificial Neural Networks (ANN'95)*, (Cambridge, UK), pp. 483–488, IEE, June 1995.

[24] Michael Witbrock and Marco Zagha, "An implementation of backpropagation learning on GF11, a large SIMD parallel computer," *Parallel computing*, vol. 14, pp. 329–346, 1990.

[25] Jim Torresen and Jon Solheim, "Implementing backpropagation training on a reconfigurable computer using pipelining of the training patterns," *Proceedings of International Conference On Neural Information Processing (ICONIP'96), Hong Kong*, vol. 2, pp. 1017–1022, Springer, September 1996.

[26] Helene Paugam–Moisy, "Parallel neural computing based on neural network duplicating," *Parallel algorithms for digital image processing, computer vision and neural networks* (Ioannis Pitas, ed.), ch. 10, pp. 305–340, John Wiley & Sons, 1993.

[27] Akira Sato, "An analytical study of the momentum term in a backpropagation algorithm. In Proc. of ICANN–91," *Artificial Neural Networks* (T. Kohonen and others, eds.), vol. 1, pp. 617–622, Elsevier Science Publishers B.V. (North-Holland), 1991.

[28] Junichi Higashino and others, "Numerical analysis and adaption method for learning rate of back propagation," *Proc. of Int. Joint Conference on Neural Networks*, vol. I, (WASH.DC), pp. 627–630, 1990.

[29] Roberto Battiti, "Optimizing methods for back propagation: Automatic parameter tuning and faster convergence," *Proc. of Int. Joint Conference on Neural Networks*, vol. I, (WASH.DC), pp. 593–596, 1990.

[30] J.R. Chen and P. Mars, "Stepsize variation methods for accelerating the back propagation algorithm," *Proc. of Int. Joint Conference on Neural Networks*, vol. I, (WASH.DC), pp. 601–604, 1990.

[31] T.P. Vogl and others, "Accelerating the convergence of the back propagation method," *Biological Cybernetics*, vol. 59, pp. 257–263, 1988.

[32] H. Kitano, "Emperical studies on the speed of convergence of neural network training using genetic algorithms," *Proceedings 8th JMIT national conference in artificial intelligence*, vol. 2, (Boston MASS), pp. 789–796, 1990.

[33] Michael McInerney and Atam P. Dhawan, "Use of genetic algorithms with back propagation in training of feed-forward neural networks," *Proc. of IEEE Int. Conference on Neural Networks*, pp. 203–208, 1993.

[34] S. Oliker and others, "Design architectures and training of neural networks with a distributed genetic algorithm," *Proc. of IEEE Int. Conference on Neural Networks*, pp. 199–202, 1993.

[35] Daniel F. McCaffrey and A. Ronald Gallant, "Convergence rates for single hidden layer feedforward networks," *Neural Networks*, vol. 7, no. 1, pp. 147–158, 1994.

[36] D.E. Rumelhart and G.E. Hinton and R.J. Williams, "Learning internal representation by error propagation," *Parallel Distributed Processing*, vol. 1, pp. 318–362, The MIT Press, 1986.