Analysis of speedup as function of block size and cluster size for parallel feed-forward neural networks on a Beowulf cluster

Fabian Mörchen

***Abstract*— The performance of feed-forward neural networks trained with the Backpropagation algorithm on a dedicated Beowulf cluster is analyzed. The concept of training set parallelism is applied. A new model for run time and speedup prediction is developed. With the model the speedup and efficiency of one iteration of the neural networks can be estimated as a function of block size and cluster size. The model is applied to three example problems representing different applications and network architectures. The estimation of the model has a higher accuracy than traditional methods for run time estimation and can be efficiently calculated. Experiments show that speedup of one iteration does not necessarily translate to a shorter training time towards a given error level. To overcome this problem a heuristic extension to training set parallelism called weight averaging is developed.**

The results show that training in parallel should only be done on clusters with high performance network connections or a multiprocessor machine. A rule of thumb is given for how much network performance of the cluster is needed to achieve speedup of the training time for a neural network.

***Index Terms*— backpropagation, parallel, training set parallelism, beowulf cluster, run time prediction, speedup, weight averaging**

I. Introduction

Neural networks, feed-forward networks with supervised learning in particular, have long since evolved into a standard data processing tool. The Backpropagation training algorithm for multilayer networks described by Rumelhart, Hinton and Williams (1986) [12] was a breakthrough at that time. It provided the groundwork for many improved algorithms with faster convergence like RPROP [11] and QuickProp [4], more commonly used today.

One problem often encountered is the extensive training required of a network. It can take a very long time for a network to converge to a sufficiently small error. Also, to build a good model, neural networks often need to be trained with large amounts of input data. Last but not least, many parameters of the model are found in practice by trial and error. Even worse, most parameters are not independent, but influence each other. Thus, for a given data set, many different parameter sets need to be tested by training a network. An obvious idea is to speed up the training by parallelizing it. The structure of feed-forward networks is inherently parallel. A lot of research has been done in the early 90's on parallelizing the training, but for the most part this work is concerned with special hardware. Work has been done on parallel neural networks either on general purpose SIMD (Single Instruction Multiple Data) parallel computers like the MasPar MP-1, hypercube machines, connection machines (CM-1, CM-2, CM-5), or even on hardware specialized in neurocomputing like the DREAM (dynamically reconfigurable extended array multiprocessor) or systolic array designs (see [14] for an overview).

An interesting architecture for high performance computing is the Beowulf cluster design [18]. First introduced in 1994 at NASA, it has become increasingly popular as an alternative to supercomputers. The idea to build clusters with several off the shelf base systems interconnected via an Ethernet network spread throughout the science community. Usually the computers are dedicated for the computation tasks. Compared to an integrated supercomputer, a Beowulf cluster has a great disadvantage: the speed of communication. Transferring data from one processing unit to another has to be done over network connections while many dedicated supercomputers have shared memory or high bandwidth internal connections between the processors. In the case of older clusters, this will only be an Ethernet connection with 100MBit or even 10MBit. Modern cluster setups are addressing this problem with higher network speeds above 1GBit or with optical technology above 10GBit [22]. But processor speed is also improving at the same time and it is the ratio of processing speed and communication bandwidth that determines the ability of a cluster to speedup non trivial problems [2].

There are different ways to parallelize a feed-forward neural net (training set parallelism, neuron parallelism, synapse parallelism, pipelining [14]), but none of them is trivially parallelizable [3], in that a significant amount of communication is needed. The parallel approach on a cluster will definitely suffer by slow communication; yet, the benefit of clusters, the possibility to use many processors in parallel, motivate one to pursue the possibilities. Several different parallelizations have successfully been implemented on special hardware, as mentioned above. Implementations on a Beowulf cluster include a systems that tries to simulate biologically accurate neurons [10] and a general modular system for neural networks [8]. Aberdeen *et. al.* [1] discuss Backpropagation performance on a Beowulf cluster, but they concentrate on performance optimization for Pentium III processors. This paper aims to explore ways of parallelizing feed-forward neural networks trained with Backpropagation on a cluster of personal computers (PC). Does it make sense to train in

Fabian Mörchen, Data Bionics, Philipps-University Marburg, Hans-Meerwein-Strasse, 35032 Marburg, Germany (email: fabian@mybytes.de)

parallel on a Beowulf cluster? Which parallel approach should be picked? Under what conditions can a speedup be observed? How many computers should be used for good efficiency?

The rest of this paper is organized as follows. In Section II we introduce methods of parallelization and performance measures for parallel computing. Training set parallelism will turn out to be the best approach for this parallel architecture. The hardware and software used are described briefly in Section III followed by the test problems in Section IV that represent different applications and network architectures. In the main part of the paper in Section V a model for accurate run time prediction will be developed and verified. It will serve to estimate the performance of a given neural network with different block sizes and cluster sizes. The practical implications of this prediction are explored in Section VI. In Section VII an apparently new extension to training set parallelism is developed to overcome the problem of slow convergence encountered with large block sizes. Finally a rule of thumb for the ratio of processor speed to network speed needed to achieve speedup is given.

II. Parallelization

Feed-forward networks have an inherently parallel structure. Every neuron does calculations independently from all other neurons in the same layer. But, during the forward pass, the input of a neuron depends on all neurons in the previous layer, and during the backward pass, the local gradient of a neuron depends on all local gradients of the succeeding layer. A good introduction to neural networks can be found in [5] and with an emphasis on practical issues in [4].

The Backpropagation algorithm is a true gradient descent method if the weight updates are accumulated and applied after one complete iteration through all patterns. This method is called *learning by epoch* and can be shown to converge to a local minimum [12]. In practice, the weight updates are often applied directly after calculating them for each pattern. This is called *learning by pattern*. As long as the learning rate is small, this method usually still converges, and for many applications this is even a lot faster. A compromise is a strategy known as *learning by block*, where the weight updates are applied after the presentation of several patterns (called a *block*) to the network. Training set parallelism can only be used with learning by block or in the extreme case with learning by epoch. Note, that modern variants of the Backpropagation algorithm with faster convergence usually learn by epoch. They are thus not considered in this study because learning by epoch is trivially parallelizable with training set parallelism.

The easiest way to speed up the training of several networks with different parameters to find good values for the learning rate, etc., is called *training session parallelism*. Each computer trains a network with given parameters and all the input data. Based on the performances of these networks, the next set of parameters is chosen, and the process can be repeated. This is trivially implementable, but it does not speed up the training of a single network, which is also desirable. Decreasing the time span between choosing a set of parameters and getting the result, e.g. of the classification performance for these parameters, would be helpful, especially because networks may take hours or days to train.

Training set parallelism divides up the training set. Each processor still has a complete copy of the network but does the training only with part of the data. Here, the weight updates need to be communicated. This strategy will benefit from large weight update intervals. Note that the minimum block size is the number of processors. It is explored in detail in Section V.

A different approach is to divide the network and not the data. This is called *node parallelism*. The first subtype is called *neuron parallelism* or *vertical slicing*. Here, each processor does the calculations for a subset of the neurons from each layer. For the forward pass, the weights of all the incoming connections for these neurons are stored locally. After each layer, the processors need to exchange the calculated outputs of the neurons in this layer to advance to the next layer. More problems arise during the backward pass. Now the weights of all outgoing connections are needed to calculate the local gradient. This is done either by calculating partial products on each processor and communicating the results to find the sum or by storing these weights locally as well, doing the necessary calculations on each node. Note that this requires more communication on part of the weight updates.

The second subtype is called *synapse parallelism* or *horizontal slicing*. Here, the problem is simply reversed. All outgoing weights are stored locally; thus, partial products with communication of the results are needed for the forward pass, and the backward pass can be easier. Some implementations use neuron parallelism in the first layer and synapse parallelism in the second layer. But, all these approaches have in common that communication of intermediate results or the output of the neurons is needed in between each layer during the forward pass as well as the backward pass. This amounts to a lot more communication than for the training set parallelism where even in the worst case of learning by pattern, communication is only needed after one complete forward and backward pass of the network. For architectures with slow communication, this is not well suited; it has been successfully implemented on Transputers [14] and other architectures that offer faster communications.

A method with similar disadvantages is *pipelining*. Here, each processor calculates the output of all neurons in a layer and passes the results to the next processor that calculates the next layer. One processor takes care of the weight updates. A technique used here that can also be applied in different contexts is *delayed weight update*. While the error for one pattern has not yet been calculated, the start of the pipeline is already processing the next pattern with the same set of weights. As soon as the weight updates are available, they are applied. The forward and backward pass are, thus, not clearly separated anymore, but interleaved.

When looking for the best way of parallelization, one should also keep in mind the inherent limitations on the number of processors that can be used efficiently. More processors will not always be beneficial. Training set parallelism scales with the number of patterns, while node parallelism scales with the size of the network, and pipelining scales with the number of layers. This means that pipelining should be used for networks

with many layers otherwise only few processors can be used. So, independently from the argument of slow communications, training set parallelism seems most promising because large amounts of input data are more common than large networks.

The measure for the gain of a parallel program over the sequential version, the *speedup* [9], is defined by

$$S(n,p) = \frac{T_\sigma(n)}{T_\pi(n,p)} \tag{1}$$

where n is the problem size and p the number of processors used. $T_\sigma(n)$ is the run time of the fastest known sequential program run on one processor of the parallel system. $T_\pi(n,p)$ is the run time of the parallel program run on p processors. The case $S(n,p) = p$ is called *linear speedup* and is rarely achieved because a parallel program usually has some overhead like the communication needed between the processors. Another measure of how well a program scales is the efficiency

$$E(n,p) = \frac{S(n,p)}{p} \tag{2}$$

An efficiency of 1 corresponds to linear speedup, while $E(n,p) < \frac{1}{p}$ indicates slowdown [9].

III. Technology

The program to simulate feed-forward neural networks with Backpropagation was written in the C programming language in combination with the GNU Scientific Library (GSL) [20] for numerical computing. The GNU Compiler Collection [19], with versions 2.91.66 and 2.96 was used. The library used for parallel programming is the Message Passing Interface (MPI) in form of the MPICH implementation [21] versions 1.2.1 and 1.2.3.

The tests were run using Linux on two different clusters (Table I) maintained in the Department of Mathematical Sciences at the University of Wisconsin-Milwaukee. Both had a topology with two stacked switches, the memory size is given per node.

TABLE I
Cluster specifications

Cluster	Nodes	Processor/Memory	Ethernet/Switch
Bat Cave	8	PIII/933MHz 256MB	100MBit D-Link DSS-8+
Messner	22	P4/1.9GHz 512MB	1GBit 3Com 4900

IV. Test Problems

The main focus of this paper is the speedup of training, thus only minimal effort was put into the parameter selection and network architecture. All networks have at most two hidden layers, even though more layers can perform better for some problems. The learning rate, the momentum rate and the block size with the best convergence were chosen after a few tests for each network. The following three different tests problems, called *Function*, *Encoder*, and *Digits* subsequently, were used.

The first problem, used to test the neural network implementation was a function simulation. The task was to learn the nonlinear function $f(x,y) = \frac{1}{\pi}\left(\sin(10x)+1\right)\arctan(-x) + \frac{1}{\pi}\left(\cos(10y)-1\right)\arctan(-y) + \frac{1}{2}$. The training set consisted of 1024 points, the test set held another 128 points. Uniform noise from the range $[-0.05, 0.05]$ was added to introduce the network to the problem of over-fitting. The domain was restricted to $[0,1]^2$. From initial tests on a sequential machine the learning rate $\eta = 0.9$ and momentum rate $\alpha = 0.5$ on a 128-128 network were chosen for further experiments. Block sizes of 4, 8, and 16 showed acceptable performance, while a block size of 1 or larger block sizes had slower convergence.

The Encoder problem is commonly used as a benchmark for neural nets [12]. The task is to find an encoding for the input data that is shorter and then decode it to get the original input back. The input data consists of all orthonormal base vectors in $\mathbb{R}^N$ where N is the number of input neurons. The network has one hidden layer with $log_2(N)$ neurons. The output layer has the same size as the input layer, and the expected output is identical to the input. Since the performance on data outside of the training set is of minor interest when compressing data, the data was not divided into a training and test set. Note that the number of input patterns is determined by the network size. The combination $\eta = 2.0$ and $\alpha = 0.9$ on a 256-8-256 network was chosen for further experiments. Again, small block sizes worked best.

The last test problem was handwritten digit recognition. The images were taken from the MNIST Database maintained by LeCun [7]. The database offers 70,000 images of handwritten digits with a resolution of 28×28 pixels and 256 gray levels. Since the focus is on network speed and not the actual error rate of recognition, no feature extraction or great care in optimizing the network architecture was done. Note that feature extraction can reduce the number of inputs needed and improve the overall performance at the same time. The network architectures used had one input for each pixel, which produces a large input layer with 784 neurons. The hidden layers for the model are smaller than the input layer to achieve data reduction and feature extraction. The output layer with 10 output neurons is the smallest layer. The expected outputs were the orthonormal base vectors of $\mathbb{R}^{10}$, one for each digit. The parameters were chosen $\eta = 1.5$ and $\alpha = 0.9$ for a 64-64 network as the best compromise for all block sizes. The block size 16 produced the fastest convergence.

V. Training Set Parallelism

The approach in training set parallelism is straightforward. Each processor is working with a complete copy of the neural network; that is, each stores all layers and weights. The weights are initialized randomly on one processor and then distributed to the others to ensure the same starting conditions. Parallelism is achieved by dividing up the training set (and the test set) among the processors. Let p be the cluster size, n be the size of the training set, m be the size of the test set, b be the block size, i.e. the number of training patterns processed between weights updates, and for later considerations let w be the total number of weights. Then each processor only works with $\frac{n+m}{p}$ input patterns and can train the network independently from the others with $\frac{b}{p}$ patterns until a weight update is necessary. The locally accumulated weight updates are then summed up over all processors and applied to each

copy of the weights. This means that for the next set of b patterns, the weights are identical on all processors again, just like in the sequential version. The collective communication of weights is the key factor in achieving speedup because it is the only additional operation that is needed compared to the sequential version.

A. Performance Model

A mathematical model will be constructed to predict how much speedup can be expected from this parallel algorithm with different parameter settings and on a particular hardware setup. Obviously, the speedup of training set parallelism depends highly on the block size and the speed of the processing nodes as well as the speed of the network connections. The workstations are assumed to be dedicated for the computation task.

Performance analysis for sequential algorithms is usually done in an asymptotic manner with the $O()$ notation [9]. When looking at parallelization and speedup, we need to be more exact. Often, the program is decomposed into small steps like multiplication of two numbers or the evaluation of the exponential function [13]. Estimation is done with constants for these operations. The actual values have to be determined for the underlying hardware architecture with special test programs. Tests with this approach did not produce good estimates on the available hardware, though. Errors of over 20% were common. A single operation, e.g. the evaluation of the exponential function, taken out of context, did not seem to perform in the same way as inside the algorithm, which is probably caused by compiler and CPU optimizations. Here, a simpler and more accurate approach was chosen. The actual implementation was used for the measurement by adding timing functions that measure the wall clock time of the main building blocks of the algorithm and are removed for the actual training later. The timing functions were integrated into the program right before and after the blocks to measure, to have minimal overhead by the timing itself. Note that it suffices to run the program for a particular test case for a few iterations to estimate the constants; we do not need to perform a complete training of the network.

It suffices to predict the time for one iteration, because the total training time depends on the number of iterations needed to meet the stopping criterion, which is application dependent. The analysis will be done for a fixed network architecture. First, we look at the sequential run time on one machine of the parallel system. Let T_f be the time needed for the forward pass with this network and T_b the time needed for the backward pass. Let T_u be the time needed for the weight update with the momentum method, that is, for scaling and applying the last weight updates, applying the current weight updates and setting the variables used to sum up the weight changes from all patterns in a block back to zero. For the sequential version of the program, we need an additional step: the current weight updates have to be saved in a different buffer to preserve them until the next weight update for the momentum method. This is not explicitly needed for the parallel version because collecting and summing up the weight updates from all processors automatically requires a second buffer and takes care of this step. For the sequential program, let T_m be the time needed to copy all weights into a different buffer. Finally, let T_s be the time needed to calculate the sum of the squares of the errors for each pattern in the test set that is needed to calculate the root mean squared error (RMSE) later. The run times of these five different parts of the algorithm need to be measured and will be approximately constant for a given network architecture and hardware. The time needed to calculate the square root only appears once per iteration; the same applies for the time needed to check the stopping criterion at the end of each iteration. Let T_o be the overhead time caused by the neglected operations, memory access, the timing functions itself, operating system overhead, etc.. The time constants for each test case on a single processor of the clusters are listed in Table II and Table III. Only the mean values are given because all variances were between 3 and 11 orders of magnitudes smaller. These constants, along with the sizes of the training set, the test set and the network (see Table IV), are the building blocks of the proposed runtime estimation.

TABLE II
TIME CONSTANTS IN MS FOR THE BAT CAVE CLUSTER

Example	T_f	T_b	T_u	T_m	T_s	T_o
Function	0.22	0.41	0.07	0.04	0.001	30.9
Encoder	0.12	0.16	0.01	0.01	0.023	28.4
Digits	1.37	1.33	0.66	0.22	0.003	314 529.2

TABLE III
TIME CONSTANTS IN MS FOR THE MESSNER CLUSTER

Example	T_f	T_b	T_u	T_m	T_s	T_o
Function	0.12	0.20	0.03	0.003	0.002	17.7
Encoder	0.07	0.09	0.01	0.001	0.013	9.4
Digits	0.26	0.27	0.92	0.058	0.003	40 218.1

TABLE IV
PROBLEM SIZE CONSTANTS

Example	n	m	w
Function	1024	128	16 768
Encoder	256	0	4 096
Digits	60 000	10 000	54 912

With these time constants, we can estimate the time needed for one iteration of the algorithm. But, we need to know how many times these actions are performed during one iteration. The forward pass has to be executed for all patterns, so the corresponding constant is multiplied by $n+m$. The backward pass is only needed for the training set; thus, the coefficient is n. Assuming that $b|n$, the number of the weight updates during one iteration is $\frac{n}{b}$. Finally, the factor for the sum of squares is the size of the test set m. Altogether, the run time for one iteration of the sequential algorithm can be estimated as shown in (3).

$$T_\sigma = (n+m)T_f + nT_b + \frac{n}{b}(T_u + T_m) + mT_s + T_o \quad (3)$$

From Tables II, III, and IV we can see that the first two summands will be dominant among the variable terms.

T_s is two orders of magnitude smaller than T_f and T_b for the function and digit example and one order of magnitude smaller for the encoder example. T_u and T_m are also about 10 times smaller. Also the first two summands contain n which will usually be much larger than m. Note that the run time decreases with increasing b because fewer weight updates are necessary, but this gain will be small. Also note that the overhead T_o is quite large for the digit example. For the function and encoder example T_o is smaller than the first two terms with $n+m$ as small as 100, but still significant. For the digits example the overhead dominates the whole term even with the large sample size present. This means that speeding up the digits example will be hard work, because more than half of the run time is constant. This enormous overhead is likely to be caused by memory access, because this example has a far larger network and larger dataset than the other examples.

If more processors are used, the training set, as well as the test set, are split up among the nodes, so the summands corresponding to the time constants T_f, T_b, and T_s will be reduced by the factor $\frac{1}{p}$, assuming both n and m are multiples of p. But on the other hand, the weight update will include a collective communication operation. All local weight updates, calculated with a subset of the training set, need to be summed up and distributed to all processors. The most efficient way to calculate a sum on all nodes, in a cluster with point to point communications, is the Recursive Doubling Algorithm [17]. Partial sums are repeatedly interchanged between 2 nodes in a tree like structure until each node has the sum of all values[1]. This tree will have $\lceil log_2(p) \rceil$ levels or time steps. For efficient communication, the number of processors should be a power of 2. The time needed for this operation was measured using a separate test program. Vectors of various length filled with random numbers were summed up using different cluster sizes. With the data collected, a linear model was built. A model with the independent variable $w\lceil log_2(p) \rceil$ and a constant term fitted the observed timings well. Figure 1 shows the data and the fitted model for the cluster Messner. The four lines are the linear approximation functions for 2, 4, 8, and 16 processors. The constant term in the model is needed because of the overhead needed to start a communication. Once the communication path is set up, the time needed for sending data from one processor to another is linear in the number of weights w. For further theoretical considerations, let

$$T_r(p) = T_w \; w \; \lceil log_2(p) \rceil + T_i \tag{4}$$

be the function for the time needed to perform the reduction of the weights over all processors, where T_w and T_i are constant. Note, that T_i includes the time for MPI initialization, the underlying TCP/IP initialization, and finally the hardware latency. It would be interesting to investigate these costs individually, because for very large clusters the overhead might not be constant. This is beyond the scope of this paper, though, and the estimation worked find with the cluster sizes used.

The estimated values for these constants for the two clusters are listed in Table V. Note that the performance of the communication call on the cluster Messner with a 1GBit network connection is not 10 times as fast as on the Bat Cave cluster with 100MBit. This is due to bandwidth limitation on the motherboards of the computers used. Note also that on Messner data can be transferred between the nodes about 4 times as fast, but the cost of initialization of a communication link is significantly higher (again, this could be due to differences in the MPI version, the TCP/IP stack or the hardware).

[1]This is implemented in the MPI function `MPI_Allreduce`

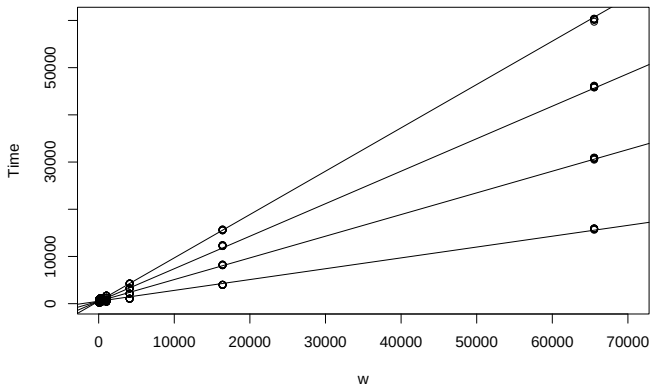


Fig. 1. Time needed for Recursive Doubling Algorithm with 2-16 processors on Messner

TABLE V
TIME CONSTANTS IN MS FOR COMMUNICATION ESTIMATION FUNCTIONS

Cluster	T_i	T_w
Bat Cave	365.868	0.798
Messner	515.250	0.230

With the function describing the time needed for the exchanging of the weight updates, the run time for one iteration of the parallel algorithm can now be estimated as shown in (5).

$$T_\pi = \frac{1}{p}(n+m)T_f + \frac{1}{p}nT_b + \frac{n}{b}(T_u + T_r(p)) + \frac{1}{p}mT_s + T_o \tag{5}$$

We can rewrite Equation (5) as follows

$$T_\pi = \frac{1}{p}\left((n+m)T_f + nT_b + mT_s\right) + \frac{n}{b}(T_r(p) + T_u) + T_o \tag{6}$$

The first term includes all calculations that profit from more processors perfectly in the sense that, if we neglect the remaining terms as well as the time needed for weight updates in (3), we would get linear speedup. However, we do have to account for the communication. Consider a given application with a fixed block size b that has the best convergence. Then, the second term gets bigger when more processors are used, but, fortunately, only proportional to $\lceil log_2(p) \rceil$. The last term is constant for fixed b. So, here we might ask the question: When does the gain in the forward and backward passes outweigh the overhead needed for the communications of the weights? How many processors can be used efficiently?

Inserting (3) and (5) into the definition of speedup in (1), we can calculate the expected speedup for a fixed problem

size as shown in (7).

$$S(p) = \frac{(n+m)T_f + nT_b + \frac{n}{b}(T_u + T_m) + mT_s + T_o}{\frac{1}{p}\left((n+m)T_f + nT_b + mT_s\right) + \frac{n}{b}T_r(p) + \frac{n}{b}T_u + T_o} \quad (7)$$

The expected efficiency is found analogous in (8).

$$E(p) = \frac{(n+m)T_f + nT_b + \frac{n}{b}(T_u + T_m) + mT_s + T_o}{(n+m)T_f + nT_b + mT_s + p\frac{n}{b}T_r(p) + p\frac{n}{b}T_u + pT_o} \quad (8)$$

B. Model Verification

To verify the proposed model for run time prediction, it was tested with one network of each example problem on both clusters. In order to make the predictions, the time constants were estimated for each test case by running it on a single processor of the clusters for a few iterations and by running a small test program to estimate the time constants for communication.

Finally the problems were run with various block sizes and cluster sizes to measure the real run times for one iteration of the algorithm. On each of the cluster sizes 1, 2, 4, 8 (and 16 for Messner) block sizes between 16 and 1024 for the function example, between 4 and 128 for the encoder example, and between 16 and 512 for the digit example were used. The models were build with the same parameters and the predicted time was compared to the observed time.

The mean error of the run time predictions made with the model described in Section V-A was roughly 5% for all examples on both clusters (see Table VI), the standard deviation was also about 5% for all settings. Apart from a few outliers all test run errors were below 10%. The model performed much better than using timings for single arithmetic operations. We will use it for further predictions of speedup and efficiency with (7).

TABLE VI
MEAN PREDICTION ERRORS OF MODEL

Example	Bat Cave	Messner
Function	4.26%	2.87%
Encoder	3.97%	5.94%
Digits	5.87%	5.73%

C. Performance Prediction

The model for run time prediction can be used to predict the speedup and efficiency for a given application and network depending on the block size and the cluster size. This way, optimal values for the size of the weight update interval and the number of processors can be chosen before starting the real training. If the block size for an application is fixed, the prediction model shows how much speedup can be achieved with a certain cluster size. Even though the cluster size is usually given in practice, using as many processors as possible might not always be the fastest choice. The model also offers the possibility to predict the performance for more processors than actually present to see whether using more would make sense.

First, the expected speedup and efficiency were calculated for the function example on cluster Bat Cave. Up to a block size of 32, the time for one iteration was larger for several processors than for one processor. With a block size of 64, the two processor setup outperforms one processor slightly; with a block size of 128 and higher, using one processor is the slowest method. Table VII lists the corresponding speedup values. These results show that a speedup is only achieved with large block sizes.

TABLE VII
PREDICTED SPEEDUPS FOR THE FUNCTION EXAMPLE ON BAT CAVE

Block size	$p = 2$	$p = 4$	$p = 8$
16	0.57	0.36	0.26
32	0.87	0.64	0.48
64	1.20	1.09	0.88
128	1.48	1.67	1.54

The speedup functions is plotted in Figure 2 with a log scale on the block sizes. With block sizes up to almost 50 no speedup will be observed; the network will run slower with several processors than on one machine of the parallel system.

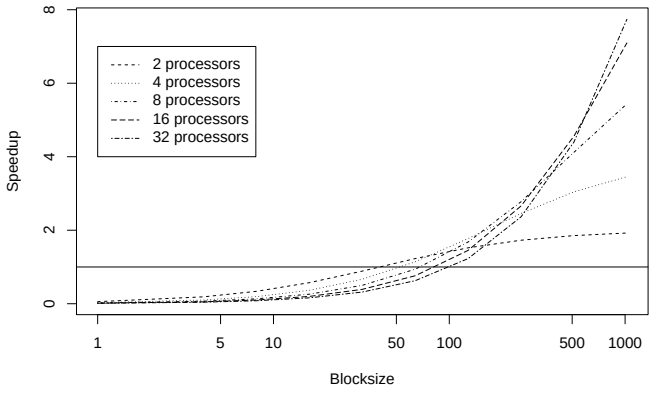


Fig. 2. Predicted speedup for the function example on Bat Cave

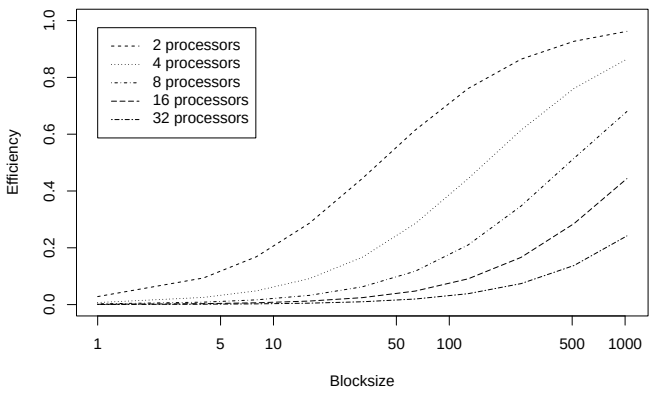


Fig. 3. Fabian Mörchen

Between a block size of 50 and 100 all graphs rise above the critical speedup value of 1. The larger the block sizes are, the better it is to use more processors. While the graph for 2 processors is best in the beginning, it is outperformed by more processors one after the other. First, it is passed by the graph for 4 processors, but this in turn is outperformed by 8 processors soon, and so on. Finally, for the largest plotted block size of 1024, which corresponds to learning by epoch with this problem, using 32 processors will be the fastest method. This illustrates that using more processors is not necessarily faster; it depends on the block size used. For small block sizes, using only one processor is fastest. For larger block sizes, more processors will run faster, exactly how many can be found using this model.

With 2 processors, we get close to a linear speedup of 2 with a block size of 128 and above. With 4 processors, a speedup close to 4 will only be achieved with a block size of 512 or 1024. For more processors, the results are even further away from linear speedup; with 32 processors, we can hardly reach a speedup of 8. This effect is also demonstrated by the efficiency plot in Figure 3. Clearly, using 2 processors is the most efficient way in this example. Independent of the block size, the efficiency is always lower for more processors. For each number of processors, the efficiency rises with larger block sizes though, because less weight updates are needed.

The Encoder example analyzed on the same cluster shows very similar behavior (see Figure 4). The graph for two processors is again the first to achieve a speedup with a block size below 20. At a block size of 50, all cluster sizes achieve speedup. Again, the larger the block sizes are, the better it is to use more processors. While two processors give almost linear speedup when learning by epoch, 4 processors reach only a speedup of 3 and 32 processors merely 6.

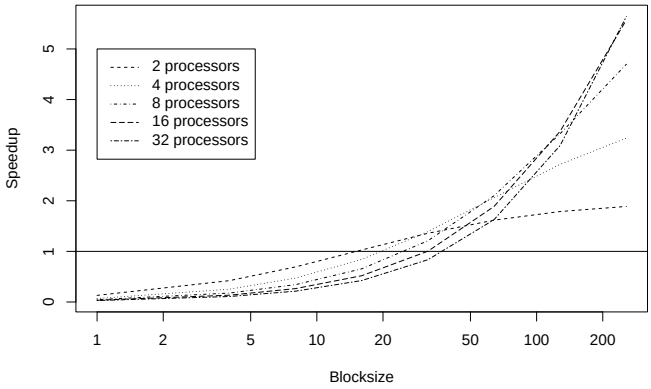


Fig. 4. Predicted speedup for the Encoder example on Bat Cave

The predicted speedup for the digit example on the cluster Bat Cave is plotted in Figure 5. The crossing of the graphs over the speedup value of 1 takes place analogously to the other examples in a small range around a block size of 50. Smaller cluster sizes have better speedup for smaller block sizes while the reverse is true for larger block sizes. But, there are two important differences to be noted in this example: A fairly good speedup can be achieved with a block size of 1000 that is small, relative to the training set size. Two and four processors produce near linear speedup, while 32 processors still have a speedup of 30. Secondly, it can be seen that the curves for the speedup flatten out for large block sizes. The block sizes do not have to be chosen very large with respect to the training set size to gain good speedup, e.g. a block size above 1000 does not produce significantly more speedup for 2, 4, and 8 processors anymore.

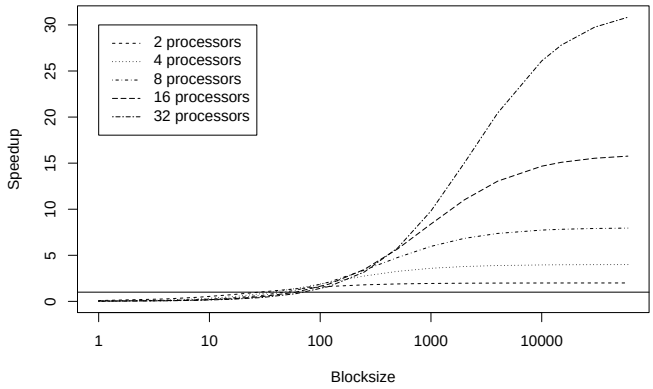


Fig. 5. Predicted speedup for the digits example on Bat Cave

This high efficiency even with many processors is achieved because the network of this example is quite large. Processing one pattern takes much longer. The more work one processor has to do before communications takes place, the better the efficiency will be.

All three example were also analyzed on the faster Messner cluster. The characteristics of the resulting plots were very similar. For the function example and the Encoder example, speedup can be achieved with smaller block sizes than on Bat Cave; that is, the graphs for all cluster sizes rise above the speedup line earlier. Messner also achieves much higher maximum speedups in these two examples, e.g. for the function example with 32 processors a speedup of 11 can be reached instead of only 8 with Bat Cave. This advantage is clearly a result of the higher communication speed. The plots for the digits example however look almost identical; the graphs cross the speedup line in about the same interval and the maximum speedup values are almost the same. In fact, the Bat Cave cluster seems to be slightly better because, in contrast to Messner, all graphs are already a little above the line speedup at a block size of 100. The reason for this is that the calculation times for this example are speeded up by a factor of 5 compared to a factor of only around 2 for the two other examples. Table VIII shows the ratio of the times needed for the forward and backward pass of the two clusters. Recalling that communications are about 4 times faster on Messner, this means that the ratio of calculation performance to communication performance for the two clusters is about the same in case of the digits example. Thus, even though the actual performance is faster, the achieved speedups are the same. A possible explanation for the higher improvement in

calculation time in this memory intensive example could be a better cache performance of the Pentium 4 processor compared with the Pentium III.

TABLE VIII
TIME CONSTANTS OF MESSNER COMPARED TO BAT CAVE

Example	T_f	T_b
Function	1.76	2.04
Encoder	1.70	1.71
Digits	5.13	4.93

VI. A PRACTICAL POINT OF VIEW

We have seen that speedup of one iteration of the training process is possible by choosing a large enough block size. How large depends on the hardware setup, the network speed in particular, and the application. In general, fairly large block sizes need to be chosen for all examples shown to achieve a good speedup on the clusters used. Considering the slower convergence for larger block sizes, it might not be beneficial to use more than one processor at all. On the other hand, the best block size in terms of convergence is often not a hard limit. Slightly larger block sizes converge slower, but usually, they can reach the same minimum on the error surface. This means that in order to achieve a speedup in the total training time to a given error level, we need to find a block size experimentally that achieves good speedup for one iteration of the training while not slowing down the convergence too much. This has to be done for each application individually.

For the function example, a small block size was beneficial for the convergence speed of the network. On one processor, we can afford to have a small block size so the reference for convergence speed is one processor and a block size of 4. For more processors, larger block sizes have to be chosen to have some speedup at all. The results for a test with block size 64 on Messner are shown in Figure 6. The reference line for one processor is shown as a solid line, as in all following plots. It is clear that no real speedup is achieved.

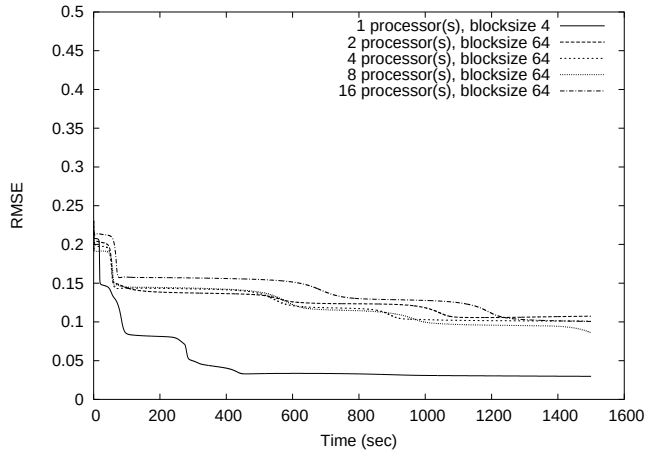


Fig. 6. RMSE of function example with 1-16 processors and block size 64

This shows that in this application the speedups shown earlier were only of technical nature. The convergence with fixed block sizes may be faster with more processors, but it is still slower than running a small block size on one processor. The slowdown in convergence caused by less frequent weight updates outweighs the speedup achieved by dividing the training set.

For the Encoder problem, a real speedup was observed. In Figure 7, you can see the development of the RMSE for one processor with block size 4 compared to 2, 4, and 8 processors on the Bat Cave cluster with a block size of 64. After about 1,700 seconds, the 8 processor curve shows the lowest error. After 3,000 seconds, the 4 processor test run also catches up with the sequential performance. Similar effects were observed on Messner; here, an even smaller block size of 32 was used for the multiprocessor tests. Using 16 processors outperforms the sequential test already after 200 seconds; the curves for 8 and 4 processors follow later.

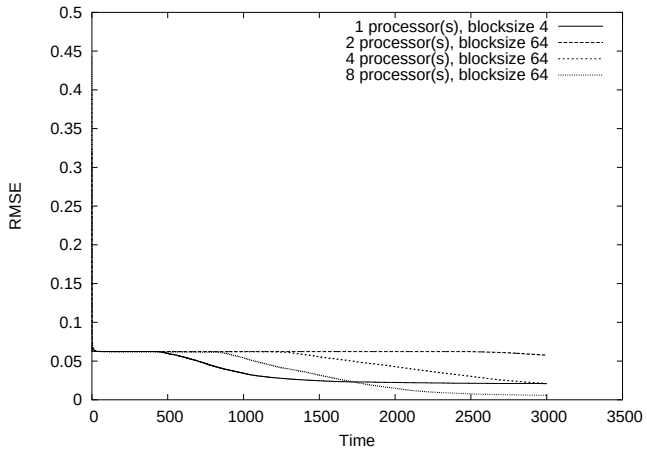


Fig. 7. RMSE of Encoder example with 1-8 processors and block size 64

In this example, the slowdown in convergence is being compensated by the speedup of one iteration. The trade off works; we can actually train the network faster when using more than one processor.

For the digits example, the results look negative again, at first. In Figure 8, you can see the comparison of block size 16 run on one processor of Messner and block size 200 with up to 16 processors. The errors on the test set were plotted instead of the RMSE because this number is more relevant for the application. Note that even without feature extraction and network optimization a recognition rate of 97% was reached. The big drop from the high error level to the low error level comes later when run in parallel than for the sequential version. Even though the speedup for one iteration is higher, the slower convergence of the network is dominant in the beginning. A closer look at the end of the training phase reveals that using the cluster can be of advantage if fine tuning of the network is required. Figure 9 shows an enlarged view of the last 500 seconds. While the single processor training achieves a total of 323 errors on the test set, the multi processor training is better for all cluster sizes, the 4 processor setup reaches the lowest test set error of 238, this corresponds to an improvement of the classification error by about 1%.

Whether the speedup for one iteration of the training

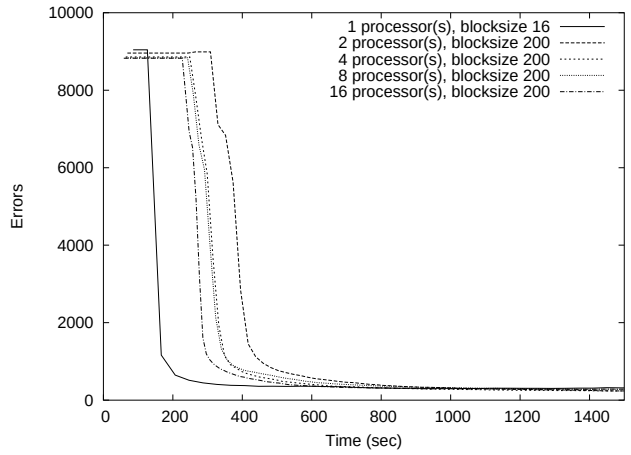


Fig. 8. RMSE of digits example with 1-16 processors and block size 200

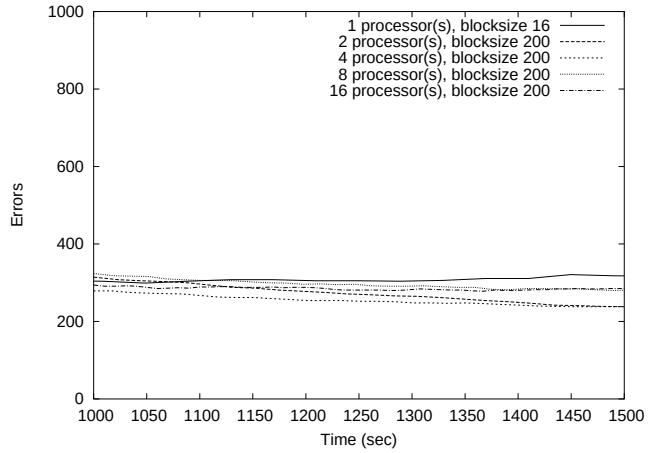


Fig. 9. Zoom into Figure 8

actually produces a speedup in the training time up to a desired error level highly depends on the convergence speed of the network chosen. The convergence speed of the function example seems to suffer severely from larger block sizes - no speedup was achieved with the clusters used here. The Encoder example, however, did show clear speedup. For the digits example a distinction between initial convergence and final convergence has to be made. The initial convergence was fastest on one processor, but later it was out run by the cluster training. Since the errors were measured on a test set and not the training set this can be counted as another successful application of the cluster setup. The initial training phase would probably also achieve speedup when using faster network technology.

VII. Weight averaging

We have seen that even though the algorithm can be speeded up in a way that more calculations are performed per second, this does not guarantee speedup in convergence. Can the problem be fixed with different approaches? The node parallelism described in Section II does not seem promising. In comparison with training set parallelism, it does offer arbitrary block sizes, but on the other hand, it requires far more communications.

A new heuristic, called *weight averaging*, was developed. The idea is to apply the weight updates calculated locally on each processor in small intervals and all updates from the other processors in large intervals. Note that this is actually a hybrid technique of training set parallelism and training session parallelism because for a short time, each processor trains the network in a different direction, independent from the others. With pure training set parallelism, all processors always have the same set of weights, which is not the case anymore with this approach. The exchange of the weight updates can be done most efficiently by averaging each weight over all processors. After all, the weights already contain the local updates. This way no buffer is needed to remember the original weights that all processors started with after the last collective communication, and no additional buffer to keep track of the accumulated weight changes locally is needed either. Thus the name *weight averaging*.

The model built for training set parallelism in Section V-A has to be slightly adjusted for this approach to predict speedup. The time for the sequential algorithm T_σ remains unchanged because the new algorithm only changes the parallel version. For the run time of the parallel version T_π, let a be the size of the weight update interval. Then we can estimate the parallel run time using (9).

$$\begin{aligned} T_\pi &= \frac{1}{p}(n+m)T_f + \frac{1}{p}nT_b + \frac{n}{b}(T_u + T_m) \\ &\quad + \frac{n}{a}(T_a + T_r(p)) + \frac{1}{p}mT_s + T_o \end{aligned} \tag{9}$$

There are two differences to the training set parallelism. First, we need to introduce a new term with the factor $\frac{n}{a}$ that accounts for the weight averaging operation. The cost function for communication, $T_r(p)$, moves from the third term, for the weight updates, into this new term. In addition, we need to measure a new time constant, T_a, for the averaging that includes scaling of all weights to calculate the average, copying the results into the weight buffer, and resetting the buffer to collect the local weight updates back to zero. Secondly, since the third term now measures the local weight updates without communications, it now also includes a memory copy operation estimated with T_m like the sequential version in (3).

But even without estimating possible speedup with the new model exactly, it is obvious that the size of the weight averaging interval should be about the same or bigger as the block size selected before. The block size for local weight updates is now independent from the number of processors and can be chosen strictly in terms of good convergence. The function example was tested with a local block size of 4 and different weight averaging intervals. An interval of 512 performed best but it still did not produce speedup in terms of training time towards a low RMSE.

The Encoder example that could already be speeded up before, performed even better with the new technique.

For the digits example the results are also much better than with the first approach. In Figure 10 you can see the convergence of the errors with a block size of 16 and a

averaging interval of 300. The curves descent earlier than with pure training set parallelism.

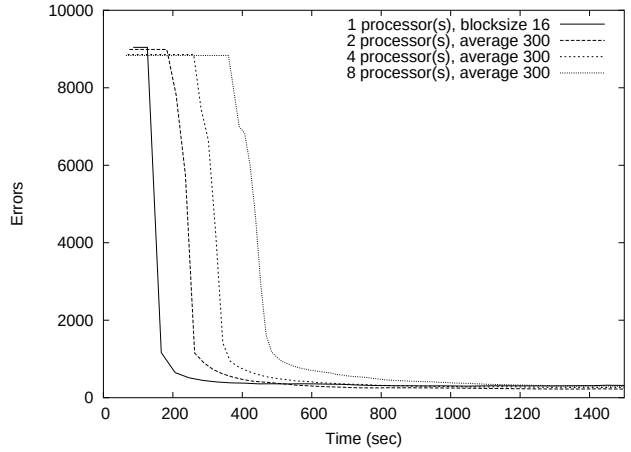


Fig. 10. Weight averaging for digits Encoder example on Messner

The convergence of the weight averaging approach was better than pure training set parallelism. The speedup of convergence for the Encoder and digit examples were better and the function example was closer to achieving speedup. This is important to notice because it means that this example could probably be speeded up with faster network technologies.

VIII. Discussion

The question whether a parallel approach is of benefit for a given problem should always be investigated carefully. If the performance gain is not significant, the additional costs for setting up a parallel system and for developing parallel programs are not worth spending.

The training of feed forward neural nets is not as easily parallelizable as it might seem at first sight. Even though many calculations can be performed independently a lot of communication is required. The training set approach was picked for parallelization of the Backpropagation training because it offers a lower communication overhead than other approaches. Another advantage of this approach is the scaling with the number of training patterns. Often, a neural network needs to be trained with a large amount of input data. On the other hand, it should be noted that the block size has to be at least as big as the cluster size. Even worse, to achieve good performance, it should be a multiple of the cluster size because otherwise, each processor trains the network only with one pattern until a weight update is necessary. Unfortunately a large block size seems to cause poor convergence in many applications, but this can be partly compensated using weight averaging.

Weight averaging effectively trains separate networks in different directions for a short time before merging the results and starting to process the next set of patterns with the same set of weights on all processors again. This approach departs even further from learning by epoch than learning by block. Even though only the latter is proven to converge to a local minimum no convergence problems were observed with the new method. This comes at no surprise because learning by pattern is widely accepted in practical applications and weight averaging bends this concept just a little further.

Building a run time model based on timings for large algorithm blocks accounts for possible code optimizations found in modern compilers. The traditional way of counting elementary arithmetic operations produces higher errors because of this optimization. It also produces much more complicated models with more terms. Measuring the building blocks of the algorithm also requires less effort because no additional test programs (except for the communication) need to be written.

IX. Conclusion

The idea of training set parallelism for neural networks on a Beowulf cluster was explored in detail in this study. Three different examples that represent different types of networks and applications were tested.

A model was built to accurately predict the run time of the algorithm for each test case with respect to different block sizes and cluster sizes. With the help of the model, a block size can be chosen large enough to achieve speedup of the time needed for one iteration. The performance for cluster sizes larger than actually available can also be estimated. The model was build in a non-traditional way based on estimated time constants for big blocks of the algorithm and for a given problem size rather than with constants for each atomic operation and with dependence of the problem size. This gave much better results because compiler optimizations and caching effects for different problem sizes can severely damage the accuracy of the latter. The time constants for the model can be efficiently estimated without a complete training of the network.

On the available hardware, fairly large block sizes had to be chosen for all examples to have speedup opposed to slowdown encountered with small block sizes. Unfortunately, large block sizes have a negative effect on the convergence speed of most neural network applications. This slower convergence was outweighed by the cluster speedup in the Encoder example and the late training phase of the digits example. The function example did not profit from being run in parallel at all. This is likely to be different with faster network connections. The following rule of thumb is proposed:

To speed up Backpropagation training of feed forward neural networks on a Beowulf cluster the ratio of processor speed (in GHz) to communication bandwidth (in MBit) should be at least 1:500.

Both clusters used in the experiments do not meet this condition: Bat Cave has a ratio of only 1:100 and Messner has 1:200. Tests run on a multiprocessor systems with a ratio of 1:500 and did show speedup of the digit recognition even in the early training phase. Unfortunately, the scaling on these systems is limited by the amount of available processors, which is typically much smaller than for Beowulf clusters. But with high performance network connections Beowulf clusters can achieve ratios even above 1:500. The rule gives a rough guidance for which hardware is needed to run Backpropagation or similar algorithms in parallel. Note, however that the speed in GHz is becoming less reliable as a performance

measure for modern processor architectures, due to special features of the chips.

The achieved speedup is also highly dependent on the application. In general, a problem has a better chance of achieving speedup if the convergence does not suffer too badly by larger block sizes. Which effect is stronger, the speedup per iteration or the slowdown in convergence, can only be found by experiment.

A new heuristic, called *weight averaging*, to compromise between low communication overhead and fast convergence was developed. The calculated weight updates are applied locally with the same block size used in the sequential version, and the set of weights is averaged over all processors in larger intervals. The results showed improvement of performance over pure training set parallelism without introducing convergence problems.

In general using algorithms with faster convergence that learn by epoch is advisable. Nevertheless our model is still applicable to those algorithms. It would be interesting to see, though, if algorithms like RPROP can profit from more frequent weight updates and weight averaging.

Future research will include parallel training of the Self-Organizing Map [6] on a Beowulf cluster. When using a large SOM with calculation of the U-Matrix [16], this is a computationally demanding task. Training set parallelism and weight averaging are promising candidates for good parallel performance of this problem.

When building a Beowulf cluster the importance of the network performance should not be underestimated. Unless dealing with trivially parallelizable problems one should rather invest more money in high performance network technologies than using the fastest processor available.

REFERENCES

[1] D. Aberdeen, J. Baxter, R. Edwards, “98c/MFLOP, Ultra-Large Scale Neural-Network Training on a PIII CLuster”.

[2] M. J. Atallah, C. L. Black, D. C. Marinescu, “Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations”, *J*ournal of Parallel and Distributed Computing 16, pp. 319-327, 1992

[3] G. C. Fox *et al.*, *Parallel Computing Works!*. Morgan Kaufmann Publishers, 1994.

[4] F. M. Ham, I. Kostanic, *Principles of Neurocomputing for Science and Engineering*. McGraw-Hill Higher Education, 2001.

[5] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Macmillan College Publishing Company Inc. IEEE Press, 1994.

[6] T Kohonen, *Self-Organizing Maps*. Springer, 1995.

[7] Y. LeCun, “MNIST handwritten digit database”, AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist

[8] C. J. G. Orellana, R. G. Caballero, H. M. G. Velasco, F. J. L. Aligue, “NeuSim: A Modular Neural Networks Simulator for Beowulf Clusters”, IWANN 2001, pp. 72-79

[9] P. S. Pacheco, *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., 1997.

[10] P. Pacheco, M. Camperi, T. Uchino, “Parallel Neurosys: A system for the simulation of very large networks of biologically accurate neurons on parallel computers.“. Neurocomputing 32-33, 1997.

[11] M. Riedmiller, H. Braun, “PROP- A fast adaptive learning algorithm”, *Technical Report (Also Proc. of ISCIS VII), Universitat Karlsruhe.*, 1992

[12] D.E. Rumelhart, G.E. Hinton, R.J. Williams, “Learning Internal Representations by Error Propagation,” *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1. MIT, 1986.

[13] N. Sundararajan, P. Saratchandran, “Parallel Implementations of Backpropagation Neural Networks on Transputers,” *Progress in Neural Processing*, Vol. 3. World Scientifi c Publishing Co. Pte. Ltd., 1996.

[14] N. Sundararajan, P. Saratchandran, *Parallel Architectures for Artificial Neural Networks*. Computer Society, 1998.

[15] S. Theodoridis, Konstantinos Koutroumbas, *Pattern Recognition*. Academic Press, 1999.

[16] A. Ultsch, “Self-organizing Neural Networks for Visualization and Classifi cation” *Information and Classification*, Berlin, Springer-Verlag, pp. 307-313.

[17] E. F. Van de Velde, *Concurrent Scientific Computing*. Springer-Verlag, 1994.

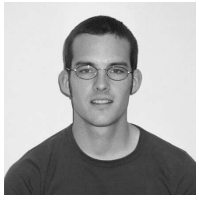
[18] Beowulf [Online]. Available: http://www.beowulf.org

[19] GNU Compiler Collection [Online]. Available: http://gcc.gnu.org

[20] GSL [Online]. Available: http://sources.redhat.com/gsl

[21] MPICH [Online]. Available: http://www-unix.mcs.anl.gov/mpi/mpich

[22] Myrinet Network Products. Myricom Inc. [Online]. Available: http://www.myri.com



Fabian Mörchen received the MS in Mathematics from the University of Wisconsin Milwaukee in 2002 and is currently writing a Ph.D. thesis on data mining and knowledge discovery in time series. He is a member of the Data Bionics Research Group at the Philipps-University Marburg, Germany.