A Low Communication Overhead Parallel Implementation of the Back-propagation Algorithm

Marcelo Alfonso, Carlos Kavka and Marcela Printista
Departamento de Informática
Universidad Nacional de San Luis
Ejército de los Andes 950
5700 - San Luis
Argentina
e-mail: {malfo,ckavka,mprinti}@unsl.edu.ar

Abstract

The back-propagation algorithm is one of the most widely used training algorithms for neural networks. The training phase of a multilayer perceptron by using this algorithm can take very long time making neural networks difficult to accept. One approach to solve this problem consists in the parallelization of the training algorithm.

There exists many different approaches, however most of them are well adapted to specialized hardware.

The idea to use a network of workstations as a general purpose parallel computer is widely accepted. However, the communication overhead imposes restrictions in the design of parallel algorithms.

In this work, we propose a parallel implementation of the back-propagation algorithm that is suitable to be applied to a network of workstations. The objective is twofold. The first goal is to increment the performance of the training phase of the algorithm with low communication overhead. The second goal is to provide a dynamic assignment of tasks to processors in order to make the best use of the computational resources.

Keywords: neural networks, parallel systems.

1 Introduction

The back-propagation algorithm is one of the most widely used algorithm for training neural networks [1]. However, it makes an intensive use of computational resources and training times of days are not exceptional. For this reason, there is a considerable interest in parallel implementations of the back-propagation algorithm.

Parallel implementation models fall in two categories: network based and training set based models.

The *network based models* partition the network components into the different processors. Each processor usually keeps a set of units and/or weights and it is responsible of the processing of these components. The processors must synchronize in order to pass the output values of each component. Usually the training is done *on-line*.

In the *training set based models* the training patterns are distributed into the different processors. Every processor processes only the patterns that are assigned to it. The synchronization is only necessary after all processors finish their forward processing cycle in order to update the weights. The training is usually done in *batch mode* which is not so efficient as the on-line mode.

Most of the parallel implementations of the back-propagation algorithm are developed for special parallel hardware [6]. Efficient implementations have been developed for parallel systems with topologies based on mesh, hypercubes, torus, lattice, systolic arrays, and using processors like FPGAs, DSPs, transputers and digital neuro-computers.

Very few implementations consider the use of interconnected workstations [7] [4] [3], which have become widely accepted as a kind of parallel computer. Plenty of computing intensive experiments in the world are carried now with a network of workstations instead of expensive special parallel computers. Just as an example, astronomic data processing and high energy experiments have selected this approach to process data. The communication overhead has now to be considered more carefully, and also the background workload has to be taken into account.

We propose a parallel implementation of the back-propagation algorithm specially designed to be used in a network of workstations where the communication overhead has to be reduced. Our proposal is a mixture of both network based parallelism and training set parallelism. The algorithm proposed is dynamic allowing a very fast and convenient reorganization of tasks between processors if it is necessary due to changes in the background workload.

The rest of this paper is organized as follows: section 2 presents a description of the back-propagation algorithm, section 3 reviews the most widely used parallel implementations, section 4 presents our proposal, section 5 discusses its advantages and section 6 presents our overall conclusions.

2 The back-propagation algorithm

A usual multilayer perceptron is shown in figure 1. The N_i input units are fully connected to the N_h hidden units, and the hidden units are fully connected to the N_o output units. This full connection pattern makes the partition of the components a difficult problem for parallel implementations.

The back-propagation learning phase consists in the following steps:

1. Initialize the weights to small random values.
2. Select a training pattern and apply it to the input units.
3. Calculate the outputs (forward propagation phase).
4. Update the weights based on the error (backward propagation phase).

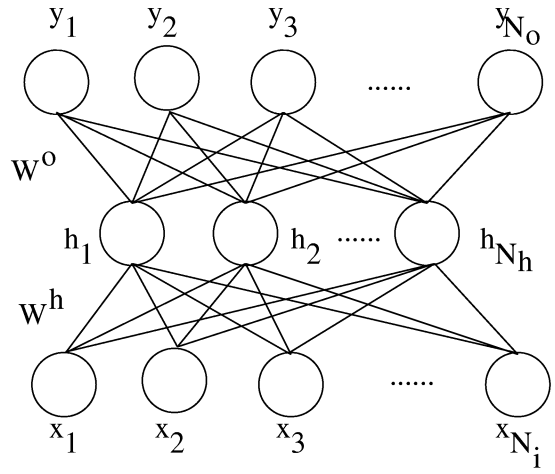

Figure 1: A multi layer perceptron

5. Repeat steps 2-4 for all training patterns.
6. Repeat steps 2-5 till the error is acceptable.

Let's define:

- input patterns: $X = x_1, x_2, ..., x_{N_i}$
- output patterns: $Y = y_1, y_2, ..., y_{N_o}$
- hidden weights: $W^h_{i,j}$ where $1 \leq i \leq N_h, 1 \leq j \leq N_i$
- output weights: $W^o_{i,j}$ where $1 \leq i \leq N_o, 1 \leq j \leq N_h$

The algorithm can be summarized as follows:

1. The activation of the hidden units (y^h_i) and the activation of the output units (y^o_i) is computed based on their inputs and the weights values:
   $y^h_i = \sum_{j=1}^{N_i} x_j * w^h_{i,j}$, where $1 \leq i \leq N_h$
   $y^o_i = \sum_{j=1}^{N_h} y^h_j * w^o_{i,j}$, where $1 \leq i \leq N_o$
2. The error in the output units (δ^o_i) and the error in the hidden units (δ^h_i) that will be used to update the weights is computed as follows:
   $\delta^o_i = y^o_i * (1 - y^o_i) * (y_i - y^o_i)$, where $1 \leq i \leq N_o$
   $\delta^h_i = y^h_i * (1 - y^h_i) * \sum_{j=1}^{N_o} \delta^o_j * W^o_{j,i}$, where $1 \leq i \leq N_h$
3. The weights in the output layer ($W^o_{i,j}$) and the weights in the hidden layer ($W^h_{i,j}$) are updated:
   $W^o_{i,j} = W^o_{i,j} + \eta \delta^o_j * y_i$, where$1 \leq i \leq N_o, 1 \leq j \leq N_h$
   $W^h_{i,j} = W^h_{i,j} + \eta \delta^h_j * x_i$, where $1 \leq i \leq N_h, 1 \leq j \leq N_i$
   When a batch learning scheme is used, the errors are collected and the weights are updated at once when the whole set of P patterns have been presented:
   $W^o_{i,j} = W^o_{i,j} + \eta \sum_{p=1}^{P} \delta^o_j(p) * y_i$, where $1 \leq i \leq N_o, 1 \leq j \leq N_h$
   $W^h_{i,j} = W^h_{i,j} + \eta \sum_{p=1}^{P} \delta^h_j(p) * x_i$, where $1 \leq i \leq N_h, 1 \leq j \leq N_i$

There exists many variations on the back-propagation algorithm that have been proposed in order to get better performance in the training time of the algorithm. They are not considered here as they do not modify the essence of the algorithm.

3 Parallel implementations

The main parallel implementations of the back-propagation algorithm fall in two categories: the training set based model and the network based model. The two following subsections provides details on them.

3.1 Training set model

In the training set model the data is partitioned and not the program. Each processor gets a subset of the training data, and performs the forward and backward propagation steps without updating the weights. Figure 2 shows how the training set can be partitioned. The δ values are collected, and when all processors finish the processing of their corresponding training data subset, values are globally collected and one weight update operation is done.

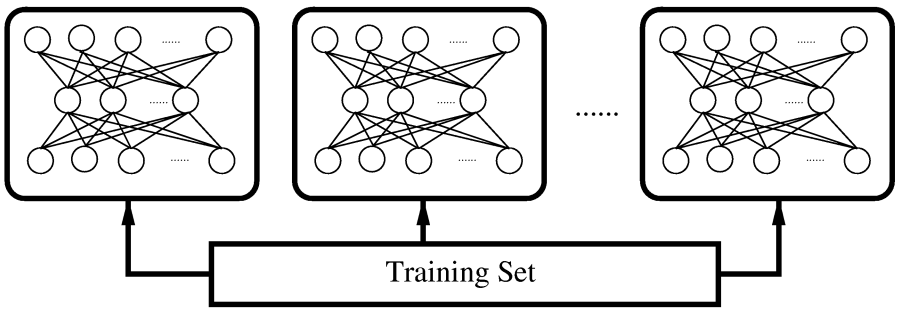

Figure 2: The training set model

Synchronization is necessary in order to update the weights in a global master processor. The data can be sent by using fixed size packets and they are only sent at the end of each training epoch.

As a disadvantage, the update in batch mode is usually less efficient than the on-line update done at the end of each training cycle.

3.2 Network based model

In the network based models, the units and/or the weights are distributed between the different processors. The model receives different names based on the distribution pattern:

- *Pipeline model*: The weights are distributed in such a way that one processor computes one layer (see figure 3). The processor that computes the lower layer has to finish before the other can start processing. However, it can process the next pattern when its output layer is been processed by the second processor. In this way, a pipeline parallelism is obtained.

- *Vertical slicing*: All incoming weights to one hidden and output unit are mapped into one processor (see figure 4). Processing can be done in parallel, but the output of the hidden units must be communicated between the processors. The weights must be sent through messages also when they are updated.

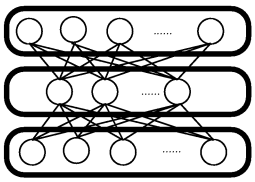

Figure 3: The pipeline model

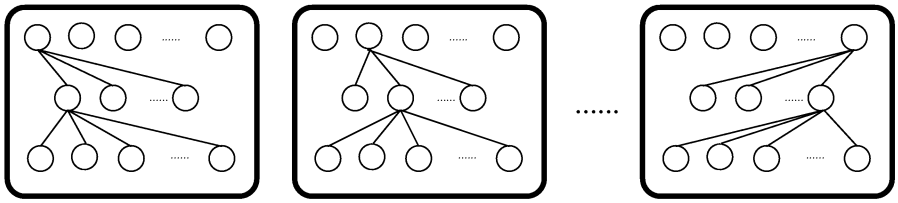

Figure 4: The vertical slicing model

- *Synapse parallelism*: The weights are distributed in such a way that each processor can compute the partial sum of each output unit (see figure 5). These results have to be added and broadcasted to the other processors. The parallelism is more fine grained.

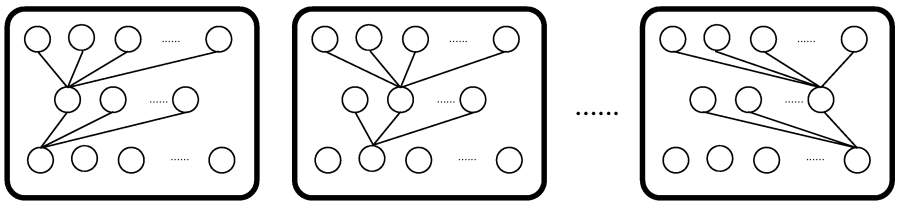

Figure 5: The synapse parallelism model

4 The proposed parallel algorithm

The target computing system is a network of workstations. It is now a day considered as a very powerful environment for high performance computing, and is becoming more and more popular. Its main characteristics are the difference in speed of the processors and the low speed communication channel between them. An algorithm to be applied in this system must match the difference in speed, reducing the idle time of some processors waiting for others. The usual practice of equal division of tasks between processors is inherently inefficient.

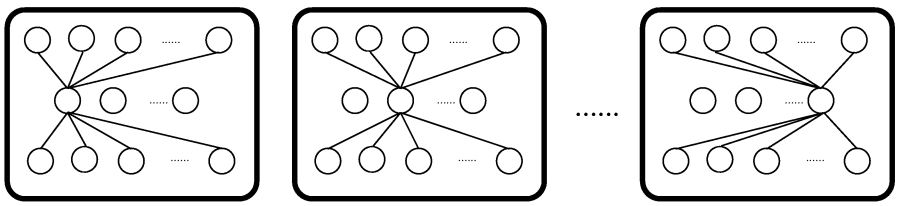

Figure 6: The proposed model

The algorithm proposed is a mixture of network based parallelism and training set parallelism (see figure 6). Each processor is responsible of the incoming and out-coming weights of a set of hidden units.

Some processors computes the forward phase of the algorithm over different patterns of the training set, and others computes the backward phase updating weights (see figure 7). The communication is reduced because there is no synchronization between them, as no processors has to wait for others. The global communication is reduced and is controlled by a parameter. The whole assignment of tasks to processors can be changed dynamically.

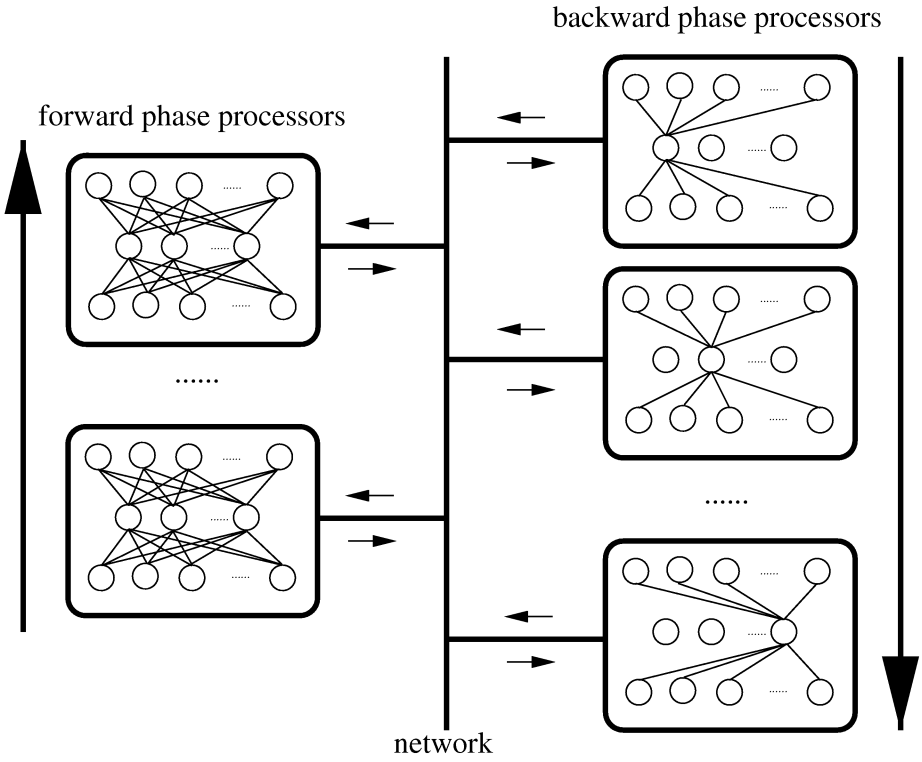

Figure 7: The task partition

4.1 The forward phase

The training set is divided in subsets by following a parallel training set model. The forward phase is implemented by a set of processors that has a copy of the network and computes the outputs and the output deltas for each pattern of the corresponding training set. The parallelism between these processors is absolute because no processor has to wait for others. The delta output and the output of the hidden nodes are sent through a packet to all processors doing the backward phase. Note that the number of values is small (if you compare with the number of weights for example), so data from a set of patterns can be collected into one message.

4.2 The backward phase

The vertical slicing approach assigns to each processor the incoming weights of some units (see figure 4). The synapse parallelism assigns to each processor the out-coming weights of some units (see figure 5). In our proposal, we assign to a processor a hidden unit (or a set of hidden units) and its incoming and out-coming weights, as it is shown in figure 6.

Each processor is responsible of these weights and it is the *only* processor that can modify them.

Every time a packet is received from a forward phase processor, only the weights under its responsibility are updated. They will be sent to the other processors only when the change in them is greater that a parameter ϵ. Note that this parameter has a direct impact on the performance of the algorithm, because it will determine the degree of communication between the nodes. See subsection 4.4 for a discussion on its values and their relation with the performance of the algorithm.

4.3 The communication

The forward phase processors send packets that contains the id of the pattern, the output deltas (δ^o) and hidden units outputs (y^h). The backward phase processors can execute a complete update cycle based just on these values, computing the hidden deltas (δ^h) and the update of the weights under their responsibility.

From the equations presented in section 2 it can be seen that each processor can update its weights based on this information in an absolute parallel way.

The backward phase processors build packets containing the weights that have changed in more than ϵ, and broadcast them to all processors.

4.4 The ϵ parameter

As it was quoted before, the selection of the value of the ϵ parameter has a direct impact on the performance of the algorithm.

- If the value of ϵ is too high, the parallelism level will be very high, because the processor will run with very few communication between them, but the performance of the training algorithm will be low, because most processors will be using outdated weight values.
- If the value of ϵ is too small, the performance of the algorithm will be near the on-line standard back-propagation algorithm, but the parallelism will be low because there will be a lot of messages going between the processors.

It is clear that a compromise selection has to be taken. The figure 8 shows the effect of different values of the ϵ parameter in a training session. The performance of the training algorithm depends on its value, but it can be seen that even with high values of ϵ, the performance of the algorithm follows quite near the standard back-propagation approach. The error change in steps because it can be measured in different iterations with the same weights, however, the effect of the error is always computed but not immediately propagated. The error reduces abruptly when the weights are updated.

The figure 9 shows the number of weights that are interchanged on average in each iteration with different values of ϵ in a standard network with 336 weights. When ϵ is 0 (default backpropagation algorithm) all the weights has to be interchanged every iteration. The communication between the processors is greatly reduced with higher values of ϵ that still produces very good training results (see figure 8). It can also be seen that the number of weights that have to be communicated changes in the different training stages of the algorithms.

It follows that the use of the ϵ parameter can reduce greatly the communication between processors without reducing the quality of the training session.

4.5 The dynamic approach

The forward phase processors operate independently on a different set of patterns. The backward phase operate on their outputs. The backward phase takes four times more time than the forward phase. It

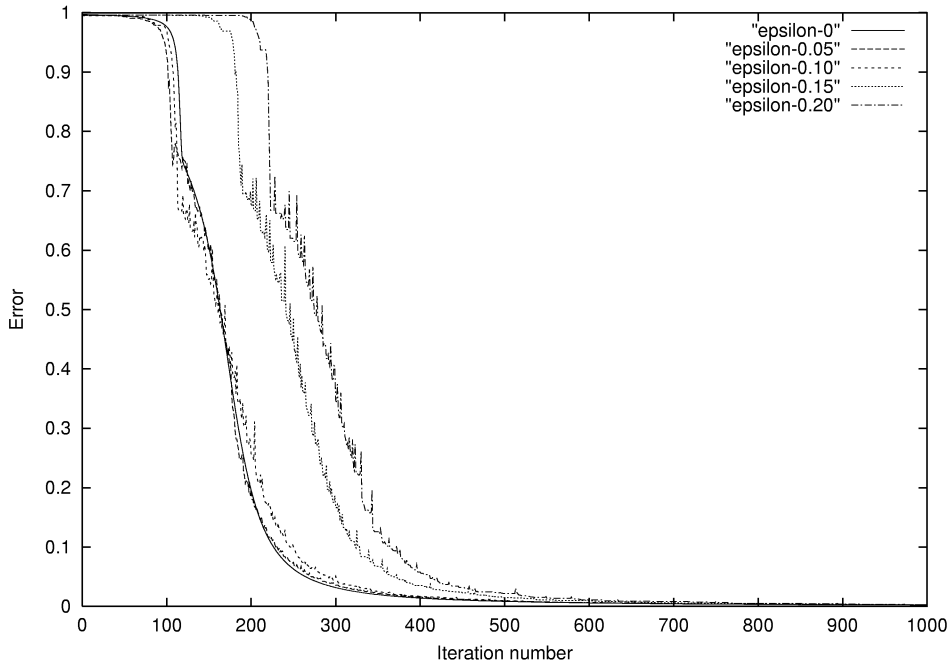

Figure 8: The effect of different values of ϵ

follows that four backward processors are necessary in order to process the data generated by one forward processor

This analysis does not consider communication overhead nor different workloads on the workstations. It is not direct to make a time diagram, like in a dedicated system or a parallel computer, to show the expected parallelism of the model. The different speed of the processors, and the different set of task involved suggest the necessity of a dynamic assignment of tasks to processors.

We implement the parallel algorithm in such a way that each processor has the same program running on it, so it can make forward phase computation or backward phase update. All processors have all the weights that defines the neural network, so it is just a matter of selecting the kind of computation that has to take place on them.

One of the processor acts like a master, and monitors the performance of the others. Two situations can arise:

- If the forward phase processors are generating data at a higher speed than the speed in which the backward processors can process it, then a forward phase processor is switched to backward mode.
- If the backward processors are idle sometime because they process the data at a higher speed than the speed in which the forward phase processor produces data, then a backward phase processor is switched to forward mode.

Note that the task switch involves just doing another task with the same data, and no program downloading takes place. The performance monitoring process is based on the length of the message queues in the backward processors.

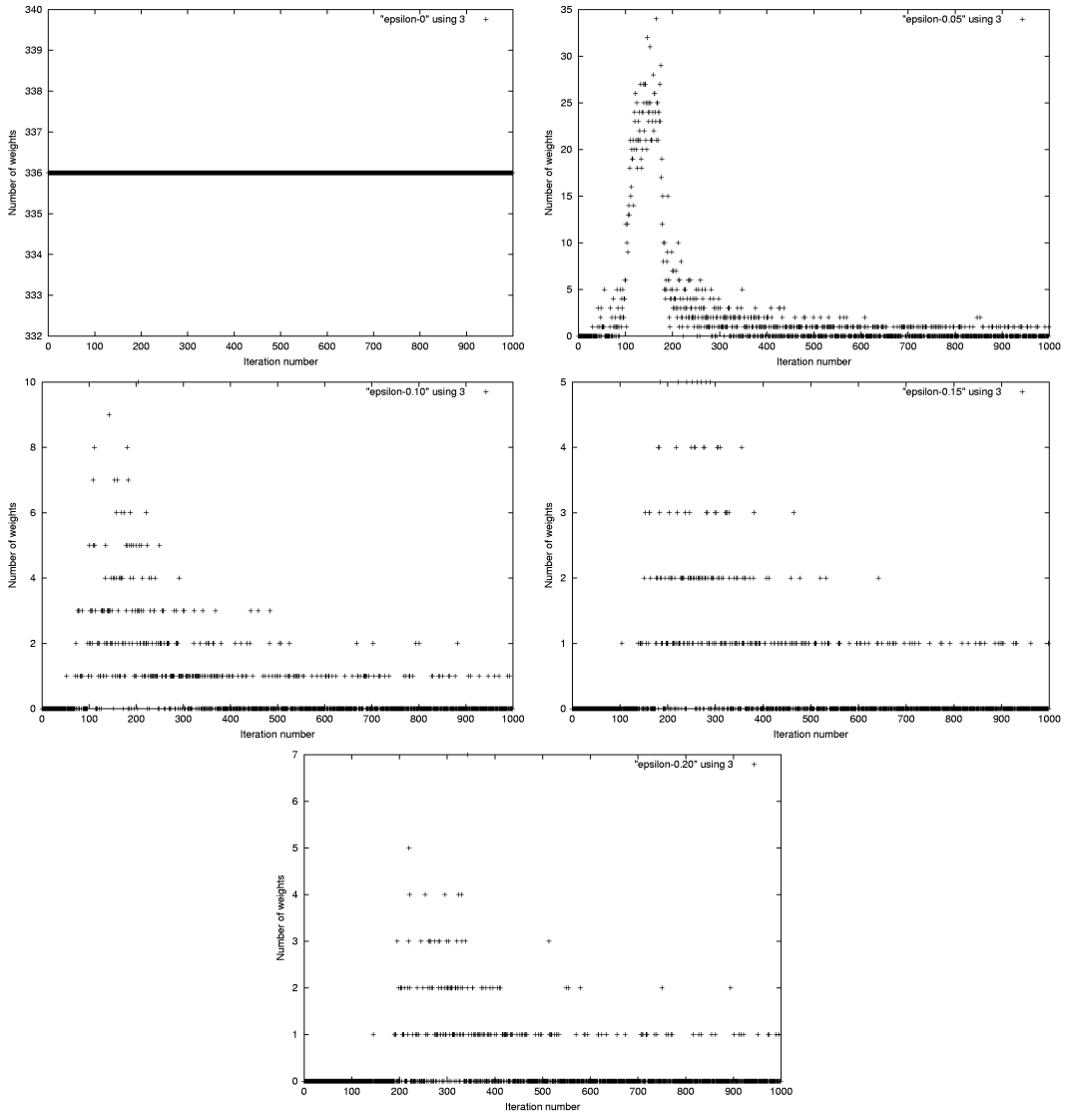

Figure 9: The effect of different values of ϵ on the number of weights that has to be interchanged

5 Discussion

This approach is a mixture of network based parallelism and training data set parallelism. The last model can be implemented efficiently only with batch learning, where network based parallelism is usually implemented with on-line learning. In training data set parallelism the communication overhead is produced at the end of a complete epoch of training. In network based parallelism, the communication overhead occurs for every pattern. In our approach, the communication overhead happens every time a forward processor processes a block of patterns, and when the weights has a major change.

Even if the patterns are processed in blocks, the backward processors can do an on-line training strategy, which is considered better than batch training. In order to reduce the communication overhead even more, the forward processors can send the summation of the errors and the backward processors can perform a block update rule, which is in between the on-line training and batch training.

The usual approach of network based parallelism (opposed to training set parallelism) would force an immediate interchange of weights, even if the values that has to be interchanged are almost the same as the originals. The ϵ approach limits the communication, but could potentially increment the error, because

some processors could use outdated values. The value of the ϵ parameter determines the degree in which this could affect the results.

The figure 9 provides an argument that shows that message interchange would be necessary in the first steps of the algorithm, but very few messages has to be send at the end. An algorithm that generates messages for every weight update (the usual approach) would generate an enormous amount of messages carrying very small modification to the weights. The loss in performance of the back-propagation algorithm due to the use of outdated values is very small at the end, and the degree of parallelism is very high because very few messages are interchanged during most of the running time of the algorithm.

The figure 9 shows how the change in the value of ϵ affects the performance of the back-propagation algorithm in different stages of the algorithm. It can be seen that the ϵ approach can be quite effective reducing the communication between the processors, while keeping an adequate level of performance for the algorithm.

The system is now being implemented on a network of Unix workstations by using the MPI (Message Passing Interface) [2] [5].

6 Conclusions

We have proposed another parallel implementation of the back-propagation algorithm for neural networks training suited to be used on interconnected workstations. It is well adapted to a system with processors with different speeds, and with dynamic changes in the workload. It implements both network parallelism and training set parallelism, with a reduced communication overhead based on a parameter that is used to determine when weights changes must be propagated. The fact that processors are responsible of some weights reduce communication, because only the modified weights must be transmitted.

7 Acknowledgments

This work was supported by the Universidad Nacional de San Luis, and the ANPCYT (Agencia Nacional para la Promoción de la Ciencia y Técnica).

The authors would like to thank to Prof. Raul Gallard, Head of our Research Group.

References

[1] E. Fiesler and R. Beale (Eds.). *Handbook of Neural Computation*. Oxford University Press and Institute of Physics Publishers, 1997.

[2] I. Foster. *Designing and Building Parallel Programs*. Addison Wesley Publishing Co., 1995.

[3] M. Printista M. Crespo, F. Piccoli and R. Gallard. A parallel approach for backpropagation learning of neural networks. *Anales del III CACIC*, 1:145–156, Septiembre 1997.

[4] M. Printista M. Crespo, F. Piccoli and R. Gallard. Parallel shaping of backpropagation neural networks in a workstation-based distributed system. *Proceedings of the International Symposium on Enginnering of Intelligent Systems - EIS 98*, 2:334–340, February 1998.

[5] M. Quinn. *Parallel Computing - Theory and Practice*. McGraw-Hill, 1994.

[6] N. Sundararajan and P. Saratchandran. *Parallel Architectures for Artificial Neural Networks - Paradigms and Implementations*. IEEE Computer Science Society, 1998.

[7] B. Wilkinson and M. Allen. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.