A PARALLEL APPROACH FOR BACKPROPAGATION LEARNING OF NEURAL NETWORKS

CRESPO, M., PICCOLI F., PRINTISTA M., GALLARD R.

Proyecto UNSL-338403[1]
Departamento de Informática
Universidad Nacional de San Luis
Ejército de los Andes 950 - Local 106
5700 - San Luis
Argentina
E-mail:{mcrespo,mpiccoli,mprinti,rgallard}@inter2.unsl.edu.ar
{mcrespo,mpiccoli,mprinti,rgallard}@unsl.edu.ar
Phone: +54 652 20823
Fax : +54 652 30224

ABSTRACT

Fast response, storage efficiency, fault tolerance and graceful degradation in face of scarce or spurious inputs make neural networks appropriate tools for Intelligent Computer Systems. But on the other hand, learning algorithms for neural networks involve CPU intensive processing and consequently great effort has been done to develop parallel implementations intended for a reduction of learning time.

Looking at both sides of the coin, this paper shows firstly two alternatives to parallelise the learning process and then an application of neural networks to computing systems. On the parallel learning side we describe main parallel schemes for a backpropagation algorithm and two alternative distributed implementations to parallelise the learning process of neural networks using a pattern partitioning approach. Under this approach, weight changes are computed concurrently, exchanged between system components and adjusted accordingly until the whole parallel learning process is completed. On the application side, some design and implementation insights to build a system where decision support for load distribution is based on a *neural network device* are shown. Incoming task allocation, as a previous step, is a fundamental service aiming for improving distributed system performance facilitating further dynamic load balancing. A neural network device inserted into the kernel of a distributed system as an intelligent tool, allows to achieve automatic allocation of execution requests under some predefined performance criteria based on resource availability and incoming process requirements.

Performance results of the parallelised approach for learning of backpropagation neural networks, are shown. This include a comparison of recall and generalisation abilities and speed-up when using either a socket interface or a Parallel Virtual Machine (PVM) interface to support parallelism.

KEYWORDS: Neutral networks, parallelised backpropagation, partitioning schemes, pattern partitioning, system architecture.

[1] The Research Group is supported by the Universidad Nacional de San Luis and the ANPCYT (National Agency to Promote Science and Technology).

1. INTRODUCTION

Neural nets learn by training, not by being programmed. Learning is the process of adjustment of the neural network to external stimuli. After learning it is expected that the network will show *recall* and *generalisation* abilities. By recall we mean the capability to recognise inputs from the training set, that is to say, those patterns presented to the network during the learning process. By generalisation we mean the ability to produce reasonable outputs associated with new inputs of the same total pattern space. These properties are attained during the slow process of learning. Many approaches to speedup the training process has been devised by means of parallelism.

The backpropagation algorithm (BP) is one of the most popular learning algorithms and many approaches to parallel implementations has been studied [5][6][9][10][12][15].

To parallelise BP either the network or the training pattern space is partitioned. In *network partitioning*, the nodes and weights of the neural network are distributed among diverse processors. Hence the computations due to node activations, node errors and weight changes are parallelised. In *pattern partitioning* the whole neural net is replicated in different processors and the weight changes due to distinct training patterns are parallelised.

This paper shows the design of two distributed supports for parallel learning of neural networks using a pattern partitioning approach. Results on speedup in learning and its impact on recall and generalisation are shown. Also a useful application of neural nets as a decisor for incoming task allocation in a distributed system is discussed.

2. BACKPROPAGATION NETWORKS

A backpropagation neural network is composed of at least three unit layers; an input, an output and one ore more hidden (intermediate) layers. Figure 1 shows a three layer BP network.

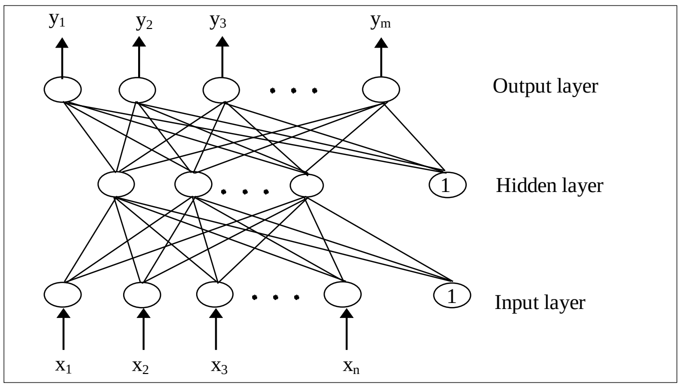

Fig. 1 - A Backpropagation Network

Given a set of p 2-tuples of vectors (x_1 y_1), (x_2 y_2), ..., (x_p y_p), which are samples of a functional mapping $y = \phi(x) : x \in R^N, y \in R^M$, the goal is to train the network in order it learns an approximation $o = y' = \phi'(x)$. The learning process is carried out by using a two phase cycle; *propagation* or *forward phase* and *adaptation* or *backward phase* [8][13][14].

Once an input pattern is applied as an excitation to the input layer, it is propagated throughout the remaining layers up to the output layer where the current output of the network is generated. This output is contrasted against the desired output and an error value is computed as a function of the outcome error in each output unit. This makes up the *forward phase*.

The learning process include adjusting of weights (adaptation) in the network in order to minimise the error function. For this reason, the error obtained is propagated back from each node of the output layer to the corresponding (contributors) nodes of the intermediate layer. However each of these intermediate units receives only a portion of the total error according to

its relative contribution to the current output. This process is repeated from one layer to the previous one until each node in the network receives an error signal describing its relative contribution to the total error. Based on this signal, weights are corrected in a proportion directly related to the error in the connected units. This makes up the *backward phase*.
During this process, as the training is progressing, nodes in the intermediate levels are organised in such a way that different nodes recognise different features of the total training space. After training, when a new input pattern is supplied, units in the hidden layers will generate active outputs if such an input pattern preserves the features they (individually) learnt. Conversely, if such an input pattern do not contain those known features then these units will be inclined to inhibit their outputs.
It has been shown that during training, backpropagation neural nets tend to develop internal relationships between units, as to categorise training data into pattern classes. This association can be either evident or not to the observer. The point here is that:

- Firstly, the net finds an internal representation, which enables it to generate appropriate responses when those patterns used during training are subsequently submitted to the network.
- Secondly, the network will classify those patterns never seen before according to the features they share (resemblance) with the training patterns.

3. Pattern partitioning for parallel backpropagation

Pattern partitioning replicates the neural net structure (units, edges and associated weights) at each processor and the training set is equally distributed among processors. Each processor performs the propagation and the adaptation phases for the local set of patterns. Also, each processor accumulates the weight changes produced by the local patterns which afterward are exchanged with other processors to update weight values.
This scheme is suitable for problems with a large set of training patterns and fit properly to run on local memory architectures [1][20].
Different training techniques, or regimes, can be implemented in a backpropagation algorithm to implement the learning process [1][10][12]. In this work, because it is appropriated for a distributed environment, the choice was the *set-training* regime. Under this technique weight changes are accumulated for all training patterns before updating any of them, with a subsequent *update all* stage each time all patterns in the training set were presented.

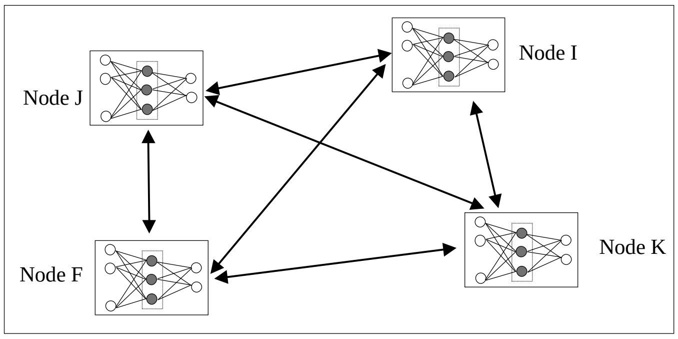

Fig. 2. A Pattern Partitioning Scheme

4. The parallel learning algorithm

To implement a pattern partitioning scheme, the whole neural network is replicated among *P* processors and each processor carries out the learning process using *Ts/P* patterns where *Ts* is the size of the training set. As shown in figure 2, weight changes are performed in parallel and then the corresponding accumulated weight changes vectors, *awc*, are exchanged between processors. Now we describe the basic steps of a backpropagation learning algorithm under the *set-training* regime, also known as *per-epoch training*[2]. The superindexes *h* and *o* identify hidden and output units respectively.

Batch-Training Algorithm

Repeat

```
    1. For each pattern $x_p = (x_{p1}, x_{p2}, ..., x_{pN})$
        1.1 Compute the output of units in the hidden layer:
            $net^h_{pj} = \sum_{i=1}^{N} w^h_{ji} x_{pi} + \Theta^h_j$
            $i_{pj} = f(net^h_j)$
        1.2 Compute the output of units in the output:
            $net^o_{pk} = \sum_{j=1}^{L} w^o_{kj} i_{pj} + \Theta^o_k$
            $o_{pk} = f(net^o_{pk})$
        1.3 Compute error terms for the units in the output layer:
            $\delta^o_{pk} = (y_{pk} - o_{pk}) f'(net^o_{pk})$
        1.4 Compute error terms for the units in the hidden layer:
            $\delta_p = f'(net_p) \sum_k \delta^o_{pk} w^o_{kj}$
        1.5 Compute weight changes in the output layer:
            $c^o_{kj} = c^o_{kj} + \eta \delta^o_{pk} i_{pj}$
        1.6 Compute weight changes in the hidden layer:
            $c^h_{ji} = c^h_{ji} + \eta \delta_{pj} x_i$
    2. Send local $c^h$ and $c^o$ and Receive remote $c^h$ y $c^o$.

    3. Update weights changes in the output layer:
            $w^o_{kj} = w^o_{kj} + (c^o_{kj}\ (local) + c^o_{kj}\ (remote))$
    4. Update weights changes in the hidden layer:
            $w^h_{ji} = w^h_{ji} + (c^h_{ji}\ (local) + c^h_{ji}\ (remote))$
Until  $(E = \sum_{=}^{T} (\frac{1}{2} \sum_{=1}^{M} (y_{pk} - o_{pk})^2) <$ maximum accepted error) or
       (number of iterations = maximum number of iterations)
```

The subindexes *p, i, j* and *k* identify the pth input pattern, the ith input unit, the jth hidden unit and the kth output unit respectively. w_{ij} is the weight corresponding to the connection between unit *j* and unit *i*, c_{ij} is the accumulated change for the weight corresponding to the connection between unit *j* and unit *i*, Θ is the bias and *N*, *L* and *M* identify the number of input, hidden and output units respectively. *f* denotes the sigmoid function which is used to compute the activation of each node.

[2] During an epoch, or batch, the submission of all patterns in the partition, the corresponding computations and the accumulation of weight changes must be performed before weights update takes place, then the next epoch begin.

5. PARALLEL LEARNING SUPPORT

Two approaches were faced for parallel learning support: a socket-based and a PVM-based architecture

5.1 A SOCKET-BASED SYSTEM ARCHITECTURE

In our early works a real implementation was built on the processors distributed in a LAN of workstations (multicomputers). Each process ran in a workstation. The routines used a socket interface as an abstraction of IPC (Interprocess Communication) mechanism
[3][4][5][6][16][17].
Figure 3 shows the support system architecture, processes and interactions. The parallel learning algorithm is independently initiated by a process at each LAN node. The *ProcW* process, will be responsible of local learning processing and when needed will request a service to exchange vectors *awc*. *ProcW* forks twice to create child processes *HI* and *HJ* for communication with other LAN nodes.

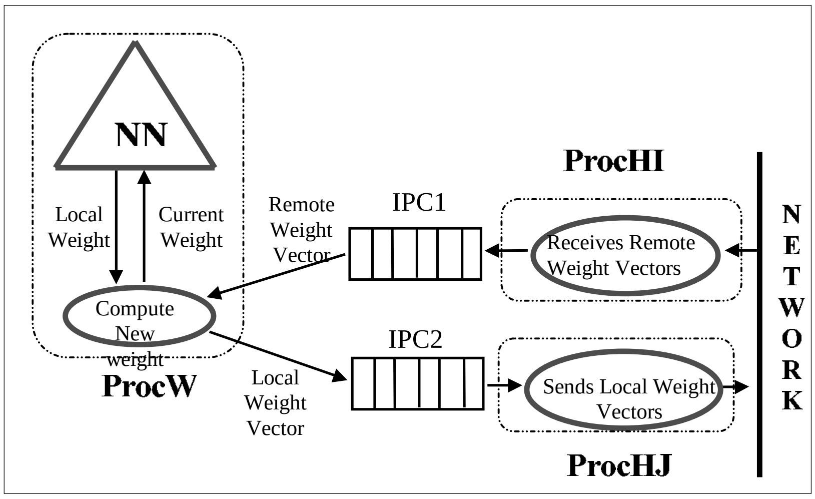

Fig. 3. - Support System Architecture

The tasks performed by each process are:

- Main process *ProcW* runs the learning algorithm for the neural net NN. After each epoch *ProcW* request the following services:
  Sending of the local *awc* vector.
  Receiving of remote *awc* vectors.
  To update the weights for every pattern in the partition it creates a current *AWC* vector by gathering the local and remote *awc* vectors.

- Process *ProcHI* receives the *awc* vectors sent by remote *ProcHJ* processes.
- Process *ProcHJ* sends the local *awc* vectors generated during one ore more epochs to the remote *ProcHI* processes.

These three processes execute independently but they communicate for exchanging information. In order to allow the concurrent execution of learning and the external communication with remote processes, a set of *non blocking* mechanisms where used.
This communication is handled via interprocess communication mechanisms (IPCs).

The interaction between processes *ProcW* and *ProcHI* is established via IPC1 when after an epoch the learning process request to distribute its *awc* vector.
The interaction between processes *ProcHJ* and *ProcW* is established via IPC2 when after an epoch the learning process request remote *awc* vectors to subsequently determine the current *AWC* vector.

5.2 A PVM-BASED SYSTEM ARCHITECTURE

The current work with PVM is discussed now. PVM is created to link computing resources and provide to the user with a parallel platform for running their computer applications, independent of the number of processors[18][19]. PVM supports a very efficient message-passing model. Figure 4 shows an alternative support to implement our particular application on the Parallel Virtual Machine.

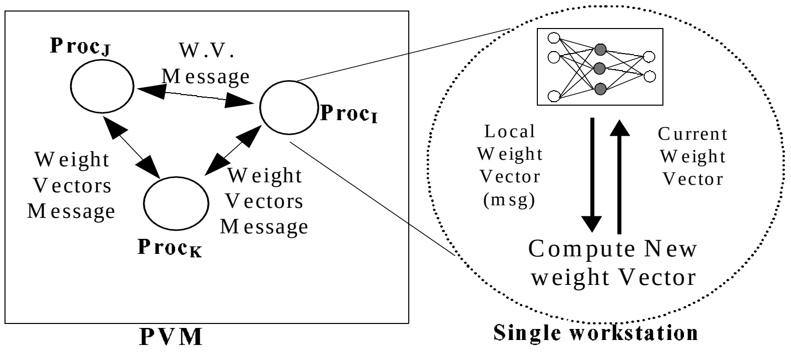

Fig. 4. System Architecture for PVM

In this section we present a brief description of the underlying system architecture and procedures supporting the parallel learning process, running on the processors distributed in a LAN of workstations. Each processor is allocated for a replicated neural network. The parallel learning support presented here is independent of the neural network structure.

5.2.1. PVM IMPLEMENTATION

The following code for BP describes the PVM implementation.

```
/*  NNBP   (parent) */
#include <pvm3.h>

void main (argc, argv )
{
        int one=1;                      /* number of  task to spawn, use always 1 for  time */
        int Task[NUMTASK];              /* children task id array */
        int i,j;

        /* Initialization of child parameters */
        for (j=1, i=NUMTASK+1; i<=argc; i++; j ++) argvnn[j] = argv[i];        /* Parameter  for each BP */
        for (j=1; j <= NUMTASK; j ++)                                          /* It spawns BP algorithm as is indicated in
NUMTASK */
        {
                argvnn[0]= argv[j];
                one= pvm_spawn("BP", (char **) argvnn, 1, PvmTaskDefault, 1, &Task[j-1]);
                if (one !=1) { pvm_exit(); return -1;}
      }
        pvm_exit();
        return -1;
}
```

```
/*  BP (child) */
#include <pvm3.h>

void main (argc, argv )
{
        int grid;                          /* group id */
        int bufid;                         /* reception buffer id  */

        grid = pvm_joingroup("NN");   if (grid<0) { pvm_exit(); return -1;}
        arguments(argc, argv);                     /* Neural Networks Initializations */

        repeat
        {
                /* Propagation Phase */

                clear_vector (&accum_weight_l) ;           /* clear weights changes vector*/
                for (each pattern)                         /* Accumulation of all weights changes*/
                {
                        /*
                          ....
                          Compute the output of units in the hidden and output  layers
                          Compute error  terms for the units in the output and  hidden layers
                          Compute weight changes in the output and hidden layers     (accum_weight_l)
                          ....
                        /*
                }

                pvm_initsend(PvmDataDefault);
                if (epoch_intervals)
                {
                        packing_weight(&accum_weight_l);  /*  packing local weight changes vector  to distribute */
                        pvm_bcast("NN", 1);
                }

                clear_vector (&accum_weight_r) ;                   /* clear weights changes vector */
                bufid=pvm_nrecv(-1,1);                             /* if bufid = 0, no message then continue */
                if (bufid>0)
                {
                        pvm_setrbuf(bufid);
                        unpacking_weight(&accum_weight_r); /* unpacking foreign weight changes vector */
                }

                update_weight(&accum_weight_l, &accum_weight_r); /* Definitive  weights changes */

                /*
                ....
                        Adaptation Phase  (back phase)
                        ....
                */

        until (current error < max. accept. err.) or  ( number of iterations ( maximum number of iterations)
}
```

Next, some comments about process interaction.
When calling PBP (on a single workstation) it will be specified as many <*file_pattern*> (pattern file names) as processors will be envolved in the training phase.
The parent spawns several BP processes with the corresponding parameters (regarding the example: BP_1 with <*file_pattern*>$_1$, BP_2 with <*file_pattern*>$_2$, etc.).
Each BP(child) will run in a different processor of the Virtual Machine.
Each child first joins at "NN" group and then individually initializes their arguments. Next, begin to run the propagation phase of algorithm on its <*file_pattern*>. When all patterns were submitted to each child, if epoch interval was reached, it broadcasts the accumulated weight changes vector and permanently receives remote accumulated weight changes vector. The reception is not blocking, since if nothing has arrived, the children must to go ahead. Finally, each BP performs the adaptation phase and completes one epoch.
Each children finish when either the current error of the neural network is less than maximum accepted error or the number of iterations is greater than the admissible number of iterations.

6. A NEURAL NETWORK DEVICE FOR ALLOCATION OF INCOMING TASKS

As an application of neural networks we used an intelligent facility to automatically allocate, in a computer network, a user incoming process to the most appropriate node in accordance to its computing requirements[2].

The model assumes that:

- The relevant performance feature to improve is the response time for user processes.
- Processes coming to be served in this network have different demands on system resources (CPU, Memory and I/O devices).
- The network is formed by a set of **N** nodes, such that each of them can contribute with different performance to a user process depending on its demands.
- Every user incoming process comes to the network through an entry node, before passing to the execution node (see Fig. 5). Process behavior and resource requirements can be determined by a program profile file or explicitly declared by the incoming process.
  - An *evaluator* module within the Operating System kernel evaluates process attributes, requirements and system state at the process arrival time.
  - Using the output of the evaluator, as input, a *decisor* module decides which node in the network can accomplish more efficiently the process execution and then process migration takes place.

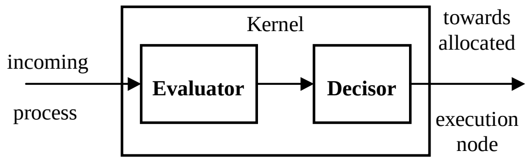

Fig. 5 The kernel portion of an entry node

As a simple example, let us assume the following scenario:

We have a system where N available nodes differ essentially in Current Memory Capacity (*CMC*) and MIPS provided. Due to system dynamics they also differ in Current Available Processing Power (*CAPP*). User processes are CPU intensive tasks and their main requirements are Memory Required (*MR*) and Desired Response Time (*DRT*).

CMC, *CAPP*, *MR* and *DRT*, are each divided into a number of levels (high, medium and low, or more levels). Other processes requirements on system resources, such as access to secondary storage, can be equally satisfied by any of the available nodes and there will not be network transfers (except for initial process migration, which we assume is equally costly for every node).

Then the following simple allocation criterion can be applied:

- Having *MR* best fit satisfied, satisfy *DRT* by allocating the process to the best fitted node (the one with minimum *CAPP* fulfilling process requirement). In case of equal *CAPP* values for more than one node then node selection is random.
- If the strategy also considers the situation where idle nodes exists, then; if for two or more nodes *CAPP* is equal, and some of these nodes is in idle[3] state (*IS*) then the process is allocated to that idle node. This second decision attempts to balance the workload.

For this allocation criterion, with N system nodes, 4+5N binary inputs will suffice to depict process requirements (4 bits) and system state (5 bits per system node), while the size of total pattern space is given by:

$$T = [\,3^{2(N+1)}2^{N}\,] - [\,3^{N+1}(2^{2N} + 2^{N})\,]$$

Because only legal inputs conform a training set for the neural net, the second term of *T* excludes the cases in which *MR* is greater than *CMC* available.

[3]A node is defined as being in an idle state when no user process is running.

7. EXPERIMENTS DESCRIPTION

Experiments covered here refer solely to the variable-epoch training regime; a new variant of the per-epoch regime. The *variable-epoch training regime* consists in randomly assigning the number of epochs (epochs interval) locally performed before any exchange takes place. Higher speed-up was the motivation of this new approach previously envisioned for a socket-based interface

The variable number of epochs locally performed before any exchange took place was chosen as a random number r between 1 and 15. Better results were observed for greater r values, but reported results corresponds to average values.

Let be *S* the total pattern space. The processors training sets (*ts*) were subsets of *S*.

For sequential training, the training set *Ts* was built by uniform selection of X% of the pattern space *S*.

For parallel training the pattern space was divided into *n* subsets and each subset was assigned to one processor (virtual or not) in parallel execution. The training subsets ts_i were built by uniform selection of *(X/n)%* of the pattern space *S*.

Values for *X* was chosen as 30 and 60.

In what follows the experiment identifiers indicate:

<Training type>-<size(%) of Training Set>/<number of subsets for parallel execution>

To compare results, the neural net was trained sequentially (i), in parallel using socket (ii) and in parallel using PVM (iii):

- (i) Experiments **SBP-X/1:** SBP-30/1 and SBP-60/1. *Size(Ts)* = 30% and 60 % of *S* respectively.

- (ii) Experiments **PBP-X/n:** PBP-30/3 and PBP-60/3. Three disjoint subsets of *(X/n)%* of *S* were selected and each subset was assigned to one processor in different workstations. Size(ts_i) = 10% and 20% of *S* respectively.

- (iii) Experiments **PVM-X/n:** PVM-30/3, PVM-30/6, PVM-60/3 and PVM-60/6**.** The software allows any numbers of processors to be created without any relationship to the number of real processors. In this state three and six disjoint subsets of *(X/n)%* of *S* were selected respectively, and each subset was assigned to one process. The number *n* of parallel processes was set to 3 and 6. *Size(ts_i)* = 10%, 5%, 20% and 10% of *S* respectively.

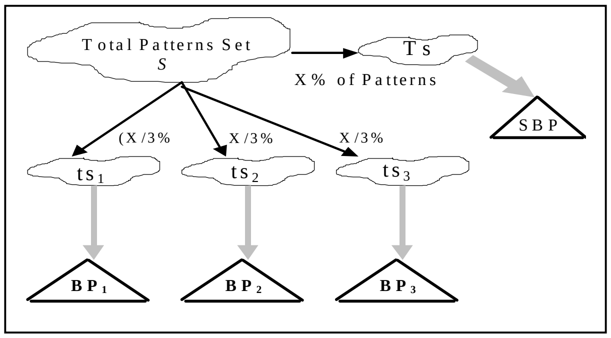

Fig. 6 – The Partitioning Approach

Figure 6 shows an example of parallel partitioning scheme for experiments SBP-X/1, PBP-X/3 and PVM-X/3.

As we were working in two stages (training and testing), the following *parameters* were used in each case:

- For Sequential Processing, *Ts* (the training set of the unique neural network) was used on the learning stage and *Ts* and *S* (the whole sample space) were used in the testing stage (for Recall and Generalisation respectively).
- For Parallel Processing, on the learning stages, ts_i was the local training subset submitted to the BP_i, with the accumulated weight changes vectors received from other BP_j networks (with training subset ts_j, $j \neq i$). For that reason,

$$Ts = \bigcup_{i=1}^{n} ts_i$$

  is the training set for all BP_i at the learning and testing stages.

During these processes, the following relevant *performance variables* were examined:

Training process:

L_T: Learning time, is the running time of the learning algorithm.

N_{iter}: Number of iterations needed to reach an acceptable error value while training.

Testing process:

R = rcg/Size(Ts). Is the recall ability of the neural net. Where *rcg* is the total number of patterns recognized when only patterns belonging to the Training Set (Ts) are presented, after learning, to the network. The objective is to analyse if each net can assimilate (can acquire) the learning of other networks that were running in parallel with it.

G = gnl/Size(*S*). Is the combined recall and generalisation ability. Where *Size(S)* is the size of the Total Pattern Space and *gnl* is the total number of patterns recognized when all possible patterns are presented, after learning, to the network.

Sp $= L_{Tapproach1} / L_{Tapproach2}$ is the ratio between the learning times under different approaches (sequential or parallel).

$Rec_{B/C}$ $= Sp/(R_{seq} - R_{par})$ is the benefit-cost ratio for recall. It indicates the benefit of speeding up the learning process, which is paid by the cost of (possibly) loosing recall ability.

$Gen_{B/C}$ $= Sp/(G_{seq} - G_{par})$ is the benefit-cost ratio for generalisation. It indicates the benefit of speeding up the learning process, which is paid by the cost of (possibly) loosing generalisation ability.

8. RESULTS

The corresponding mean values of the performance variables are shown in the following figures and tables.

As we can observe in figure 7 a reduction, greater than one third in the number of iteration needed to achieve permissible error values, was achieved. Results for the partitioning scheme of 30% are shown but using either parallel partitioning approach attains similar results.

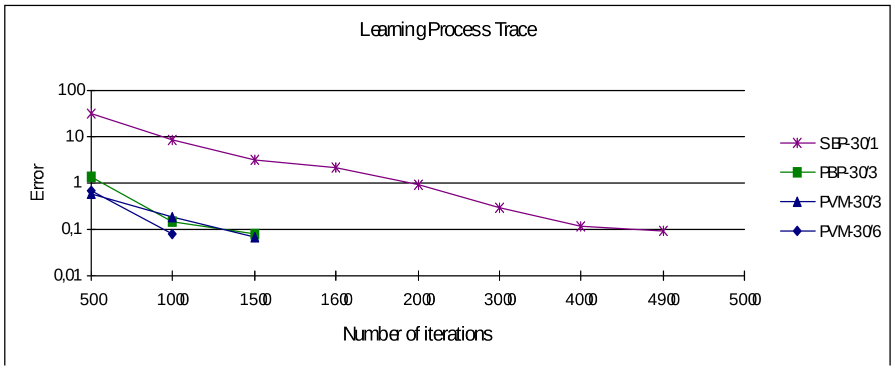

Table 1, is a summary of the experiments performed and their results:

Experiment	Learning time (seconds)	Recall %	Generalisation %
SBP-30/1	1989	100	98
PBP-30/3	426.91	97.85	96.84
PVM-30/3	123.33	97.84	95.71
PVM-30/6	53.05	92.8	92
SBP-60/1	5996	100	99.5
PBP-60/3	1137.66	99.33	98.83
PVM-60/3	275.66	99.91	99.73
PVM-60/6	87.22	98.97	97.64

Figures 8 and 9 show the associated loss in recall and generalisation of the neural network for different sizes of the training set.

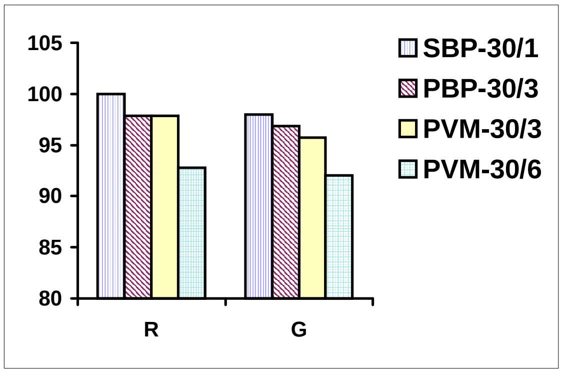

Fig. 8 - Values of Recall and Generalisation for Ts= 30 % of S

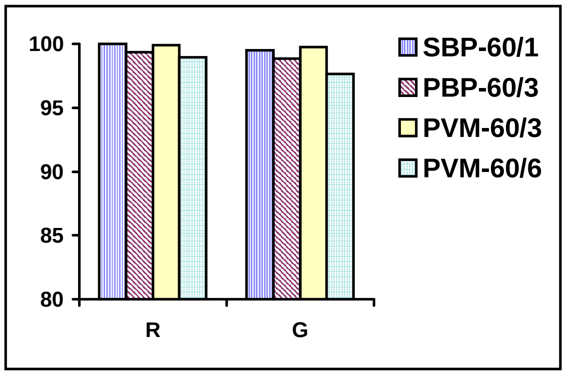

Fig. 9 - Values of Recall and Generalisation for Ts= 60% of S

In general, the detriment of recall and generalisation capabilities decreases as the training set size is incremented. Their values range from 0.09% (PVM-60/3) to 7.2% (PVM-30/6) for recall. In the case of generalisation the values range from -0.23% (PVM-60/3) to 6% (PVM-30/6). Opposite to the expected, in this case, an improvement was also detected: PVM-60/3 achieved a generalisation capability better than the sequential BP.

Table 2 indicates the speed-up in learning time attained through parallel processing when different sizes of the portions of the total pattern space *S* are selected for training the neural network. Table 2(a) shows the ratio between the sequential learning and the parallel learning times ($S_P = L_{T(S)} / L_{T(P)}$).

SBP-30/1 vs.			SBP-60/1 vs.		
PBP-30/3	PVM- 30/3	PVM-30/6	PBP-60/3	PVM- 60/3	PVM-60/6
4.65	16.12	37.49	5.27	21.75	68.74

Table 2(a) - Speed-up values achieved through parallel processing vs sequential processing

In general, as the size of *Ts* increases (from 30% to 60%) then an increment of the speed-up can be observed under variable-epoch training.
This effect shows a substantial improvement over the per-epoch approach used in earlier implementations. Moreover, increment of the speed-up can be observed among different parallel implementations. Table 2(b) shows the ratio between both parallel learning times ($S_{PVM} = L_{T(PBP)} / L_{T(PVM)}$).

PBP-30/3 vs.		PBP-60/3 vs.	
PVM- 30/3	PVM-30/6	PVM- 60/3	PVM-60/6
3.46	8.04	4.12	13.04

Table 2(b) - Speed-up values achieved through parallel processing with PVM vs. parallel processing with Socket

Both parallels implementations showed comparable capability, but PVM-X achieved a substantial increment in speed-up with values ranging from 3.46 to 13.04 times faster than PBP-X.
It is interesting to observe in table 3 the Benefit-Cost Ratio, which gives an indication of a speed-up *Sp* obtained at the cost of a detriment in recall or generalisation. Table 3(a) shows for PVM-X, the benefit-cost ratio for recall ability

$$\mathbf{Rec_{B/C}} = Sp/(R_{seq} - R_{parPVM})$$

and table 3(b) shows the benefit-cost ratio for generalisation ability

$$\mathbf{Gen_{B/C}} = Sp/(G_{seq} - G_{parPVM}).$$

SBP-30/1 vs. PVM- 30/3	SBP-30/1 vs. PVM-30/6	SBP-60/1 vs. PVM- 60/3	SBP-60/1 vs. PVM-60/6
13.96	30.29	21.66	67.71

Table 3(a) - Benefit-Cost Ratio for Recall.

SBP-30/1 vs. PVM- 30/3	SBP-30/1 vs. PVM-30/6	SBP-60/1 vs. PVM- 60/3	SBP-60/1 vs. PVM-60/6
7.03	6.24	--	37.35

Table 3(b) - Benefit-Cost Ratio for Generalisation.

In all cases, it can observed good ratios between benefit and costs. In the particular case of PVM-60/3 an increment of speed-up was simultaneously detected with an increment in generalisation capability, hence the benefit cost ratio is not registered. This performance variable is of great help to inspect the goodness of a parallel design for training neural nets.

9. CONCLUSIONS

The training time of a neural network has been a main issue of recent research. The length of a training time depends, essentially, upon the number of iterations required. On its turn, this number depends upon several interrelated factors. Some of them are: size and topology of the network, initialization of weights and the amount of training data used.
A means of training acceleration based in the last mentioned factor is a parallel approach known as pattern partitioning. Using this technique, in this work we presented a feasible architecture for a system supporting parallel learning of backpropagation neural networks on a Parallel Virtual Machine. A preliminary set of experiments in our investigation, revealed that the beneficial effects of parallel processing can be achieved with minor capability loss.
Furthermore, we need to remark that PVM provided us a unified framework within which our parallel application was developed in an efficient and clear manner. That resulted in a straightforward program structure and very simple implementation. PVM transparently manipulated all message routing, synchronization aspects, data conversion, message packing and unpacking, process group manipulation and all aspect regarding heterogeneity. All these factors contributed to reduced development and debugging time. It worth remarking that a more effective implementation of parallel backpropagation neural network was completed.

Finally, at the light of the effectiveness showed by the distributed approach for the parallel learning process by means of PVM, at the present time, testing with larger number of processors and different training set sizes are being performed for different neural networks.

10. ACKNOWLEDGEMENTS

We acknowledge the cooperation of the project group for providing new ideas and constructive criticisms. Also to the Universidad Nacional de San Luis the CONICET and the ANPCYT from which we receive continuous support.

11. REFERENCES

[1] Berman F., Snyder L. - *On mapping parallel algorithms into parallel architectures*- Parallel and Distributed Computing, pp 439-458, 1987.
[2] Cena M., Crespo M. L., Gallard R. *Transparent Remote Execution in LAHNOS by Means of a Neural Network Device*. Operating System Reviews, Vol. 29, Nro 1, ACM Press, 1995.
[3] Colouris G., Dollimore J., Kindberg T. -*Distributed Systems: Concept and Design* - Addison-Wesley, 1994.
[4] Comer, D. E., Stevens, D. L. - *Internetworking with TCP/IP* - Vol. III - Prentice Hall.

[5] Crespo M., Píccoli F., Printista M., Gallard R.- *A Parallel Approach for Backpropagation Learning of Neural Networks*- Proceedings of 3er Congreso Argentino de Ciencias de la Computación, Universidad Nacional de La Plata, Vol. 1., pp 145 – 156,, September 1997.

[6] Crespo M., Píccoli F., Printista M., Gallard R.- *Parallel Shaping of Backpropagation Neural Networks in Workstations-Based Distributed Systems*- Proceedings of International ICSC Syposium on Engineering of Intelligent Systems – EIS' 98, University of La Laguna, Vol. 2., pp 334 – 340, Tenerife, Spain, February 1998.

[7] Foster, Ian T. : *Designing and Building Parallel Programs* - Addison Wesley, 1995.

[8] Freeman, J., Skapura, D. *Neural Networks. Algorithms, Applications and Programming Techniques*. Addison-Wesley, Reading, MA, 1991.

[9] Girau, B. - *Mapping Neural Network Back-Propagation onto Parallel Computers with Computation/Communication Overlapping*. Proceedings of Euro Par'95.

[10] Kumar, V., Shekhar, S., Amin, M.- A *Scalable Parallel Formulation of the Backpropagation Algorithm for Hypercubes and Related Architectures*. IEEE transactions on Parallel and Distributed Systems, Vol. 5. Nro.10, pp 1073 - 1090, October 1994.

[11] McEntire, P. L., O'Reilly, J. G., Larson, R. E. (Editors) : *Distributed Computing: Concepts and Implementations* - Addison Wesley, 1984 .

[12] Petrowski, A., Dreyfus, G., Girault, C.- *Performance Analysis of a Pipelined Backpropagation Parallel Algorithm*. IEEE. Trasactions Networks, Vol. 4, pp 970 - 981, November 1993.

[13] Plaut, D., Nowlan, S., Hinton, G. *Experiments on Learning by Backpropagation*. Tech. Report, CMU-CS-86-126, Carniege Mellon University, Pittsburg, PA, 1986.

[14] Rumelhart, D., Hinton, G., Willams, R. *Learning Internal Representations by Error Propagation.* MIT Press, Cambridge, Massachusetts, 1986.

[15] Rumelhart, D., McClelland, J. *Parallel Distributed Processing, vol. 1 y 2*. MIT Press, Cambridge, MA, 1986.

[16] Stevens, R.W.- *Advanced Programming in the UNIX Envionment*- AdDison-Weslwy Publishing Company. 1992.

[17] Stevens, R.W.- *UNIX Network Programing.* Prentice Hall-Englewood Cliff. 1990.

[18] Sunderam, V., Manchek, R., Jiang,, W., Dongara, J., Bengeuelin, A., Geist, A. – *PVM3 – User's Guide and Reference Manual* – Oak Ridge National Laboratory: Tenesse, 1994.

[19] Sunderam, V., Manchek, R., Jiang,, W., Dongara, J., Bengeuelin, A., Geist, A. – *PVM: Parallel Virtual Machine* – The MIT Press, Cambridge, Massachusetts, 1994.

[20] Tanembaun, A. – *Modern Operating Systems* – Prentice Hall, 1992.