ELSEVIER

Neurocomputing 31 (2000) 67–85

NEUROCOMPUTING

www.elsevier.com/locate/neucom

Back-propagation learning algorithm and parallel computers: The CLEPSYDRA mapping scheme

Antonio d'Acierno

I.R.S.I.P.-C.N.R. Via P. Castellino, 111-80131 Napoli-Italy

Received 21 March 1998; accepted 29 April 1999

Abstract

This paper deals with the parallel implementation of the back-propagation of errors learning algorithm. To obtain the partitioning of the neural network on the processor network the author describes a new mapping scheme that uses a mixture of synapse parallelism, neuron parallelism and training examples parallelism (if any). The proposed mapping scheme allows to describe the back-propagation algorithm as a collection of SIMD processes, so that both SIMD and MIMD machines can be used. The main feature of the obtained parallel algorithm is the absence of point-to-point communication; in fact, for each training pattern, an all-to-one broadcasting with an associative operator (combination) and an one-to-all broadcasting (that can be both realized in $\log P$ time) are needed. A performance model is proposed and tested on a ring-connected MIMD parallel computer. Simulation results on MIMD and SIMD parallel machines are also shown and commented.

Keywords: Back-propagation; Mapping scheme; MIMD parallel computers; SIMD parallel computers

1. Preliminaries

An artificial neural network (ANN) is a parallel and distributed information processing structure consisting of a large number of Processing Elements interconnected via unidirectional signal channels called connections; its behavior is determined by parameters denoted *weights* and a learning procedure is used to compute these parameters. Such a learning procedure tends to be very time consuming and is

E-mail address: dacierno.a@irsip.na.cnr.it (A. D'Acierno)

0925-2312/00/$ - see front matter
PII: S0925-2312(99)00151-4

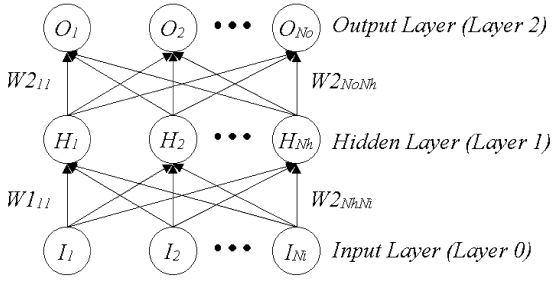

Fig. 1. A feed-forward neural network with three layers (*Ni–Nh–No*).

therefore obvious to try to develop faster learning algorithms and/or to capitalize on the intrinsic parallelism of these systems in order to speed up the computations. However, though learning algorithms typically involve only local computations, the output of an unit usually depends on the output of many other units. Thus, without careful design, an implementation of such algorithms on a massively parallel (distributed memory) computer can easily spend the majority of its running time, say 75% or more, in communication rather than in actual computation [17]. Hence, the mapping problem is not trivial and deserves attention, since the communication problem is obviously crucial.

The major motivation of this work was exactly to try and solve this problem, with reference to a well-known learning algorithm. In this paper, in fact, it is proposed a new approach to the parallel implementation of the back-propagation of errors learning algorithm (BPA) [15,16] mainly used for feed-forward ANNs; here the network topology is such that each neuron in a layer receives inputs from every node in the previous layer and sends the output only to neurons of the next layer (for an example see Fig. 1).

In the first phase of the BPA (forward phase) an input vector to the network is provided and values propagate forward through the network to compute the output vector

$$H_i = f\left(\sum_{j=1}^{Ni} W1_{ij} I_j\right), \quad i = 1 \ldots Nh, \tag{1}$$

$$O_i = f\left(\sum_{j=1}^{Nh} W2_{ij} H_j\right), \quad i = 1 \ldots No. \tag{2}$$

For the sake of simplicity, we assume that there are no biases.

The output vector $\boldsymbol{O}$ is then compared with a target vector $\boldsymbol{T}$ (provided by a *teacher*) resulting in an error vector

$$\boldsymbol{E} = \boldsymbol{T} - \boldsymbol{O}. \tag{3}$$

In the second phase (backward phase) the error vector values are propagated back through the network by defining, for each level, the error signal $\boldsymbol{\delta}$ that depends on the

derivative of the activation function. Specifically, the error signals for neurons belonging to layer i are determined from a weighted sum of error signals of layer $(i+1)$ again using the connection weights (now backward); the weighted sum is then multiplied by the derivative of the activation function. Formally, if for simplicity we suppose

$$f(x) = \frac{1}{1+\mathrm{e}^{-x}} \tag{4}$$

being

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = f(x)(1-f(x)) \tag{5}$$

we have

$$\delta_i^O = O_i(1-O_i), \quad i = 1 \ldots No \tag{6}$$

for output nodes, and

$$\delta_i^H = H_i(1-H_i)\sum_{j=1}^{No} W2_{ji}\delta_j^O, \quad i = 1 \ldots Nh \tag{7}$$

for hidden nodes. Finally, the weight changes are evaluated according to

$$\Delta W1_{ij} = \eta\delta_i^H I_j, \quad i = 1 \ldots Nh,\ j = 1 \ldots Ni, \tag{8}$$

$$\Delta W2_{ij} = \eta\delta_i^O H_j, \quad i = 1 \ldots No,\ j = 1 \ldots Nh, \tag{9}$$

where η is the learning rate (empirically chosen between 0 and 1). In the "on-line" version of the algorithm the weight changes are applied as they are evaluated while, in the "batch" or "by-epoch" version of the BPA, the weight changes are accumulated to compute a total weight change after all the (or some) training patterns have been presented.

2. Related papers

A first glance at the standard equations used to describe the BPA reveals that there are several degrees of parallelism in such an algorithm. First, if the learning is by-epoch, there is the parallel processing of many training examples (training examples parallelism, [12]). Here, on each processor a replica of the whole neural network and some training patterns are mapped; each replica evaluates partial weight changes that are then summed. Secondly, there is the parallel processing performed by the many nodes of each layer (neuron parallelism, [12]); this form of parallelism corresponds to viewing the calculations as matrix–vector products and mapping each row of the matrix onto a processor. A third parallel aspect of the batch BPA (maybe less obvious) stems from the fact that, if the learning is by-epoch, the forward and backward phases for different training patterns can be pipelined (layer forward–backward parallelism, [12]). Last, since the incoming activation values

must be multiplied by a weight, there is parallelism also at neuron level (synapse parallelism, [12]); this form of parallelism corresponds to viewing the calculations again as matrix–vector products but mapping each column of the matrix onto a processor.

Despite this, a parallel implementation of the BPA that splits the connection matrices among processors is not a very simple task since such a parallel implementation should efficiently perform the product of matrix $\|\mathbf{W2}\|$ by a vector as well as the product of the transpose of $\|\mathbf{W2}\|$ by a vector (Eqs. (2) and (7)). As a matter of fact, the papers dealing with the parallel implementation of the BPA can be grouped into three main categories. The first one groups the works in which each processor simulates one neuron or one single connection (see, for example, [6]). While this approach seems well suited when the aim is an hardware implementation, the use of general-purpose parallel computers is penalized by the amount of communication required.

In the second group we found the papers in which the learning is supposed by by-epoch and the training examples parallelism is used, i.e. a data partitioning is proposed. Witbrock and Zagha [18], for example, proposed an implementation using the data partitioning on the GF11, a SIMD parallel computer composed (in the final version) of 556 processors, each of which is capable of 20 MFLOPS. They obtained, using a machine with 356 processors, 900 MCUPS (Millions of Connections Updated per Second) on the NETTALK benchmark (203-60-26 with 12022 training patterns). Pomerleau et al. [14] implemented the batch BPA on Warp, a 10 processor system, obtaining 17 MCUPS on the NETTALK benchmark. A quite different approach has been proposed by Bourrely [3]; he implemented a modified version of the BPA on the IPSC hypercube and assumed again that each node holds a replica of the network. Each replica is trained i times in batch mode using randomly selected examples; the weight changes evaluated after the i iterations are sent to a master that calculates the mean. Such mean changes are transmitted back to nodes that update weights and the cycle restart. A speed-up of 28.92 with 32 processors is obtained with $i = 32$, while, with $i = 1$, a speed-up of 12.78 (again with 32 processors) is achieved. These solutions, although are well suited for special applications and allow very fast simulations, do not solve the problem, since high-order network cannot be simulated even if a large number of processors are available and, besides, under the assumption of batch-learning the computation may diverge. Moreover, the *neuro-scientist* should not be coerced into using the batch version of the algorithm.

The last category of papers (to writer's mind the crucial one) groups works where a network partitioning is proposed. Most of the papers cited in the following brief overview (that of course is neither complete nor exhaustive) might at a first glance seem out of date; the problem is that the author is trying to address just *seminal* papers, i.e. paper where a solution is *proposed* and not simply *implemented*.

Typically, the BPA is seen as a sequence of matrix–vector products that are usually performed in a systolic fashion. As it is well known, there are two systolic solution to the matrix–vector product problem. The first one (herein called *by-row*) is obtained by mapping, on each processor, a row of the matrix [11]. The elements of the vector circulate through the processor network; when the ith element of the vector reaches

the jth processor it is multiplied by the ith element of the jth row of the matrix; the result is added to a partial sum. After n steps (where n is the dimension of the square matrix) the computation terminates and the partial sum in the jth processor contains the jth element of the resulting vector. Since the elements of the vector are transmitted unchanged, the computation can be easily overlapped with the communication.

The second systolic algorithm for the matrix–vector product (herein called *by-col*) maps a column of the matrix and an element of the vector on each processor [10]. In this case intermediate sums are moved from one processor to another; here, of course, communication and elaboration are not overlappable.

Kerckhoffs et al. [7] observed that the problem concerning the mapping of $||\mathbf{W2}||$ can be trivially solved by storing these weights twice, i.e. they proposed a solution exploiting the parallelism among neurons belonging to the same layer and supposed that each process knows both incoming and outgoing weights of handled hidden neurons. This method, clearly, introduces an overload in the updating phase since some connections have to be modified twice. Then, they stated that the by-col method can be efficiently used in evaluating error terms of hidden neurons.

Zhang et al. [19] implemented the batch BPA on the connection machine and combined data partitioning and network partitioning. They realized the network partitioning again through the use of the by-row method to evaluate the activation values of hidden and output neurons and of the by-col method to evaluate the error terms of the hidden neurons. Then, they applied again two times the by-row method to evaluate weight changes; this choice, although saves memory space, seems very strange, since the data to be transmitted are the same as that transmitted in the forward phase. The simulator they obtained, however, is very fast; as Singer states [17], this is mainly due to (i) the fact that such a simulator avails itself of both neuron and training examples parallelism and (ii) the intimate knowledge of strengths and limitations of the connection machine hardware.

Kung and Hwang [10] used the by-row solution in the forward phase and the by-col solution for evaluating the error terms of hidden neurons. Then, they observed that such a method is efficient when the sizes of different layers is approximately equal. If this is not the case, they derived the design of a two-layers systolic machine; to be fully utilized, such a machine requires a quite complicated synchronization scheme so that the mapping of it on a general-purpose parallel computer seems neither trivial nor very efficient.

Kumar et al. [9] introduced a technique, called checkerboarding, for mapping feed-forward networks on hypercube-based machines. In this scheme, the processors form a $\sqrt{p} \times \sqrt{p}$ mesh embedded on a p-processor hypercube; nodes of each layer are equally distributed among the diagonal processors while weights are equally distributed among all processors.

Ahmed and Priyalal [1] described how the solution based on the vertical slicing can be improved through a keen analysis of the communication involved using a multiple bus system. A very interesting solution has been proposed in [8] where the parallel implementation, still based on intra-layer parallelism, exploits the collective nature of the communication involved (see also [2]).

Last, it is worth addressing the approach pursued by Petrowski et al. [13]; they implemented the batch version of the BPA on Transputer arrays formulating the batch algorithm as a sequence of matrix–matrix products and used the algorithm proposed by Fox et al. [5] to perform these products; the weight matrices as well as the state vectors are thus evenly distributed among nodes. The main problem is that weights have to be transmitted in the forward phase as well as in the backward phase and this makes the implementation communication intensive. Besides, even if the connection matrices are split among processors, the need of batch learning has been not eliminated.

3. The proposed approach

As it has been shown in the previous section, the exploitation of neuron parallelism (*vertical slicing*), eventually combined with the training examples parallelism, represents the most frequently used method to obtain fast simulations of the learning phase of feed-forward networks. The approach here proposed aims to improve the performance through the use of a mixture of neuron parallelism, synapse parallelism and training examples parallelism (if any).

With reference to the on-line BPA, suppose that

1. the memory of each processor is limitless;
2. *Nh* equals the number of processors to be used.

Having in mind these hypotheses, we can suppose that each process knows all the training patterns and handles an hidden neuron. As a first step, each process evaluates the activation value of the handled hidden neuron; suppose that process i knows the ith row of $||\mathbf{W1}||$, there is no communication and the load is perfectly balanced.

To evaluate the output value of output neurons, each process in our implementation (say process i) calculates the vector J^i representing the product of Hi and the ith column of $||\mathbf{W2}||$. Once these vectors have been evaluated (and this evaluation can be performed in a fully parallel and perfectly balanced way if process i knows column i of $||\mathbf{W2}||$), they are summed through a read, sum and send algorithm; the processes are thus logically organized as a tree rooted by a *master* process. The master completes the evaluation of the net inputs to output neurons (by subtracting biases, if any), applies the activation function and evaluates the error terms of output neurons. Such error terms are then broadcast to the other processes (see Fig. 2 for a topological view of the scheme).

Once the error terms of output neurons have been received, the error of the handled hidden neuron must be evaluated. This step only requires that process i knows the ith column of $||\mathbf{W2}||$ (Eq. (7)) to be executed; since this hypothesis is verified, there are no data to be communicated and the load is perfectly balanced.

Finally, each weight must be updated; since each process knows all it needs to evaluate the weight changes for the connections it knows (and handles), we assume again that there is no communication and the load is perfectly balanced.

To summarize, in the implementation here proposed (*Clepsydra* Mapping Scheme) it is used as a mixture of synapse parallelism and neuron parallelism so that an

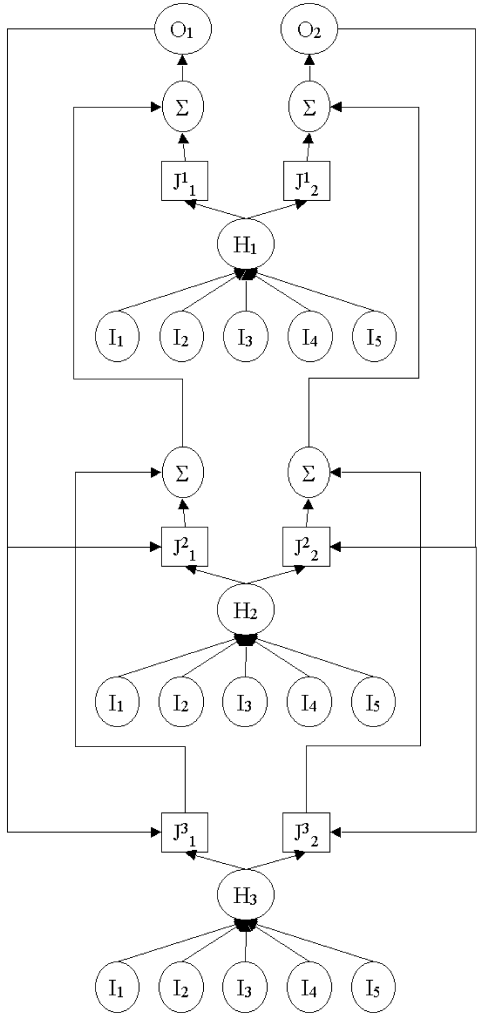

Fig. 2. The proposed mapping scheme for a network with five input neurons, three hidden neurons and two output neurons.

all-to-one broadcasting and a one-to-all broadcasting are required for each training pattern. More precisely,

- neuron parallelism is used to evaluate the activation values of hidden neurons;
- synapse parallelism is used to evaluate the activation value of output neurons;
- neuron parallelism is used to evaluate the error terms of hidden neurons.

The out-coming collection of processes works in a SIMD fashion. Since both MIMD and SIMD machines can be used and, compared with the classical vertical slicing, the method proposed here maybe presents some advantages.

First, by using the vertical slicing, the maximum number P_{vs}^{max} of processors that can be used equals the dimension of the smaller layer. When, obviously, $Nh \geq \min(Ni, Nh, No)$, the proposed method could allow the use of more processors. Such a feature makes Clepsydra (mainly) useful for problems where the output layer is much smaller than the other layers (e.g. classification problems).

Secondly, even if the number P of the available processors is less than P_{vs}^{max}, the vertical slicing requires, to assure the load balancing, that P exactly divides Ni, Nh and No. More precisely, the dimension of each layer must be typically modified (by adding *dummy* neurons) to verify such a hypothesis and this of course introduces an overload. By using the Clepsydra scheme, instead, the condition to be verified is that P exactly divides just Nh, and it is worth remembering that this is a free parameter. Besides, such a condition is strictly required only if synchronous machines are taken into account; if asynchronous machines are used the topology of the network does not need modifications so that no overload is introduced (of course the best efficiency is obtained when P exactly divides Nh, see Section 5).

Third, it is a matter of fact that parallel machines are not widely used since the algorithms must be completely re-coded since it is often not possible to re-use existing code. Regarding the problem at hand, the porting of a simulator on parallel platforms by adopting the vertical slicing does not seem a trivial task, being maybe easier to re-write the code, since communication steps are distributed across the code and data structures of sequential and parallel algorithms do not match. By using the Clepsydra scheme, such a task is not prohibitive, since each sub-network equivalent to a network with the same topology of the network to be simulated, except for the hidden layer that has Nh/P units. Moreover, the communication steps are localized and primitives for concentration and broadcasting of data are widely available.

Last, it is worth emphasizing that the proposed method requires just a all-to-one associative broadcasting and a one-to-all broadcasting while the vertical slicing requires that each processor has to communicate with each other at least two times (even supposing that each process knows all the training patterns). Although the use of systolic methods allows to perform data exchanges in a very efficient way, it is a matter of fact that (i) systolic solutions do not allow the use of powerful communication facilities that could improve performance and (ii) topologies different from the ring cannot be (easily) used. The Clepsydra mapping, instead, can be easily mapped on each topology and, because of the communication required, is able to fully exploit all communication facilities (like, for example, the *scan* functions in SIMD machines). Moreover, if some patterns can be processed in batch mode, it is possible to group data to be transmitted so that the overhead due to start-up operations can be diminished.

To conclude this section the proposed method must be clearly extended to real cases by removing hypotheses 1 and 2 we made at the beginning of this section.

The first hypothesis has been made to suppose that each process knows all training examples. In classification problems, for example, the training set is composed by two sub-sets: the first one represents the input part of training examples, while the second sub-set represents the output part. It should be clear that the output part must be stored only in the memory of the master processor so that this processor should have

more memory than other processors. As regards the input part of training examples, this set must be known by each processor; such a requirement can be often satisfied. In fact, with reference, for example, to a character recognition problem and supposing that each character is represented with a 16×16 matrix of bit (that seems a reasonable representation after that an actual pre-processing has been applied), 2 Mbytes of memory are enough to store more than 60,000 examples. Anyway, if the memory is not enough to store the whole training set, there are several possibilities. First, if the learning must be strictly on-line, we can suppose that each process knows just a maximum of Ni/P points of each training examples; the by-row method can be then applied to evaluate the activation values of hidden neurons while nothing changes in the other steps. Another suitable solution consists in storing in the memory of each processor α/P patterns, where α represents the number of training examples. Before the processing of each pattern, the latter is broadcast to all other processes (this allows to use *scan* facilities). Finally, if some patterns (say K patterns) can be processed in batch mode, it is possible to replicate the algorithm proposed above. Each processor now has to store α/K patterns, that could be feasible, if K is not too small. The possibility offered by the use of this solution (that of course combines network partitioning and data partitioning) allows to remove also Hypothesis 2. In fact, the maximum number of processors that can be used now is NhK. Clearly, since replicas of the algorithm have to combine weights, the performance increases if a number of replicas much smaller than K is used. It has to be pointed out that the combination of weight changes is performed in a parallel way, i.e. the process s of each replica has to communicate only with process s of each other replica.

4. Performance analysis

As a first step, analyze the time complexity of a sequential simulation. In the following we will indicate with T_i^a the time complexity of phase i (where i is the label of the formula in Section 2) in the sequential case ($a = S$) or in the parallel one ($a = P$). It is of course

$$T_1^S = k1 * Ni * Nh + k2 * Nh, \tag{10}$$

$$T_2^S = k1 * Nh * No + k2 * No, \tag{11}$$

$$T_3^S = k3 * No, \tag{12}$$

$$T_6^S = k4 * No, \tag{13}$$

$$T_7^S = k1 * Nh * No + k4 * No, \tag{14}$$

$$T_8^S = k5(Ni * Nh), \tag{15}$$

$$T_9^S = k5(Nh * No). \tag{16}$$

The $k\#$'s clearly represent dimensional constants. In the parallel case it is

$$T_1^P = \frac{T_1^S}{P} \tag{17}$$

since there is no communication and the load is perfectly balanced. Of course, we are supposing for simplicity that P exactly divides Nh; if this is not the case the time in Eq. (17) as well as in the following equations depends on $\lceil Nh/P \rceil$:

$$T_2^P = \frac{k1(Nh * No)}{P} + g(No,P) + k2 * No, \tag{18}$$

where $g(No, P)$ measures the time to perform an association on the chosen topology, having as data to be combined a vector of dimension No:

$$T_3^P = k3 * No, \tag{19}$$

$$T_6^P = k4 * No + g'(No, P), \tag{20}$$

where $g'(No, P)$ measures the time to perform a one-to-all broadcast, having as data to be transmitted a vector of dimension No. Last, we have

$$T_i^P = T_i^S/P, \quad i = \{7, 8, 9\}. \tag{21}$$

To summarize, we have

$$TS = \sum_i T_i^S = k1(Nh * Ni + 2 * Nh * No) + k2(Nh + No) + (k3 + 2k4)No + k5 * Nh(No + Ni). \tag{22}$$

Since

$$TP = \sum_i T_i^P = \frac{k1}{P}(Nh * Ni + 2 * Nh * No) + \frac{k2}{P} * Nh + (k2 + k3 + k4)No + \frac{k5}{P} * Nh + \frac{k5}{P} * Nh(No + Ni) + g(No, P) + g'(No, P) \tag{23}$$

we obtain

$$TP = \frac{TS}{P} + g(No, P) + g'(No, P) + \frac{P-1}{P} * No(k2 + k3 + 2k4). \tag{24}$$

It is worth noting that $k2$ is related to the evaluation of the sigmoid function, while $k3$ and $k4$ just concern additions and products; so Eq. (24) can become

$$TP \approx \frac{TS}{P} + g(No, P) + g'(No, P) + \frac{P-1}{P} * No * k2. \tag{25}$$

We have tested this performance model on a MIMD machine, namely on a ring connected array of T800 INMOS Transputers with 20 MHz of clock speed and

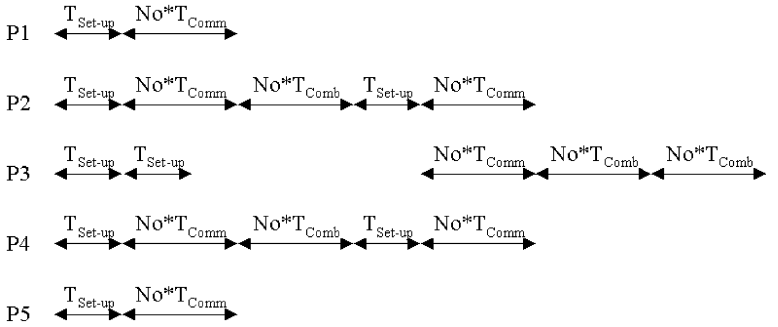

Fig. 3. The function $g(No, P)$ when P is 5.

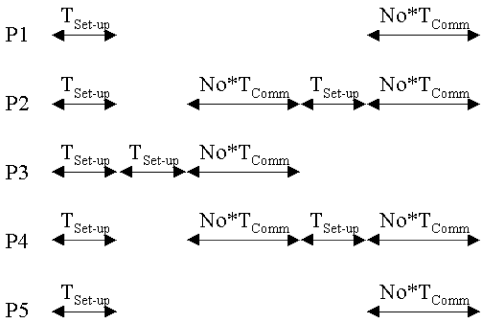

Fig. 4. The function $g'(No, P)$ when P is 5.

20 Mbit/s of link speed. In this case, calling

$T_{Set\text{-}up}$ = 1.13 μs the time to initialize a communication [4];
T_{Comm} = 4 * 1.56 μs the time to send or receive 4 bytes [4];
T_{Comb} = 2 μs the time to perform an addition (assuming a sustained performance of 0.5 MFLOPS);

we have that $g(No, P)$ can be expressed as (see Fig. 3):

$$g(No, P) = \left\lceil \frac{P-1}{2} \right\rceil (T_{Set\text{-}up} + No * T_{Comm}) + \left\lceil \frac{P+1}{2} \right\rceil T_{Comb} * No. \tag{26}$$

Similarly, $g'(No, P)$ can be expressed as (see Fig. 4):

$$g'(No, P) = \left\lceil \frac{P+1}{2} \right\rceil T_{Set\text{-}up} + \left\lceil \frac{P-1}{2} \right\rceil T_{Comm} * No. \tag{27}$$

We assumed as test case a network with 200 neurons in the input layer and 60 neurons in the hidden layer. The sequential time (in milliseconds, for an iteration on

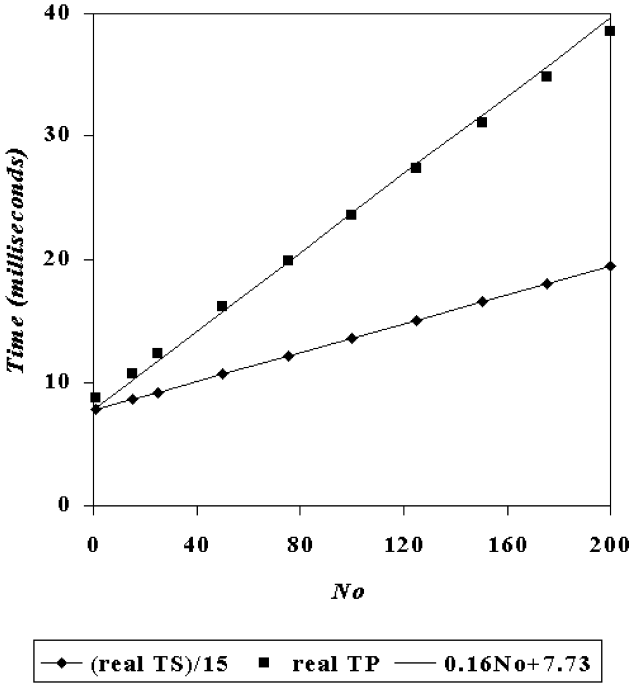

Fig. 5. Theoretical and experimental results as No varies on a ring-connected Transputer array ($P = 15, Ni = 200, Nh = 60$, learning on-line, one iteration on one example).

a single example) can be expressed as a function of No:

$$TS = 0.88No + 116. \tag{28}$$

Since it is (when P is, for example, 15)

$$g(No, P) + g'(No, P) = (103.35No + 16{,}95)\ \mu s \approx No * 10^{-1}\ ms \tag{29}$$

and by neglecting $k2 * No$ (tens of μs) we have that the theoretical time (in ms) for the simulation is

$$TP = 0.16 * No + 7.73. \tag{30}$$

Fig. 5 shows the differences between the theoretical parallel execution time and the measured one.

Finally, we fixed No (equal to 15) and measured the sequential time for a complete iteration on one example (121.1 ms). The theoretical time (in ms) for the parallel simulation as a function of P is

$$TP = \frac{121.1}{P} + 0.1 * P. \tag{31}$$

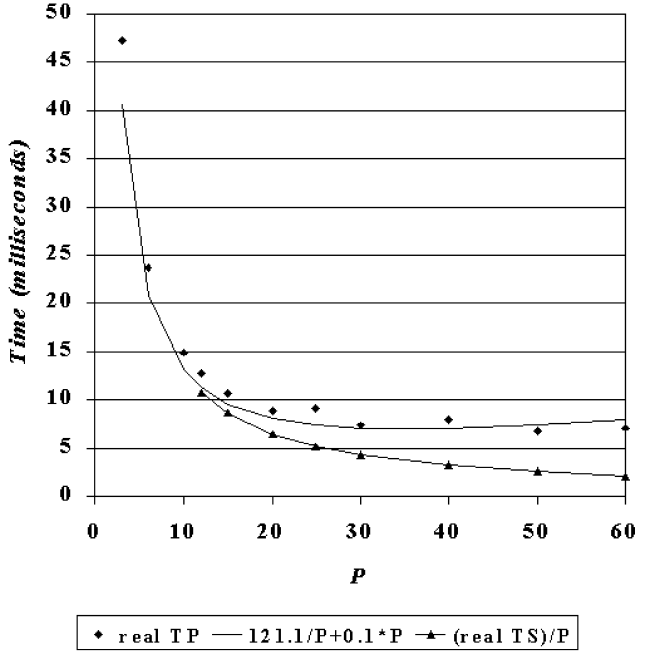

Fig. 6. Theoretical and experimental results as P varies on a ring-connected Transputer array ($Ni = 200, Nh = 60, No = 15$, learning on-line, one iteration, one example).

Fig. 6 shows the difference between the measured and theoretical simulation times; the theoretical model, clearly, works better when P exactly divides Nh.

Eq. (25), theoretically derived and whose validity has been also experimentally proved, needs two considerations. First, it must be noted that such a formula just represents a lower bound for the performance of the proposed method. In fact, when an asynchronous machine is taken into account and the learning is by-epoch at least on a few examples, the communication can be easily overlapped with the elaboration, as it will be shown in the next section. Secondly, it is worth noting that, through the use of communication facilities and/or using "good" topologies like trees and hypercubes, the functions $g(No, P)$ and $g'(No, P)$ can become of course of order $No * \log P$. Moreover, the use of a common bus for executing the one-to-all broadcast can make the function $g'(No, P)$ of order No.

5. Simulation results

In this paper are shown simulation results on a MIMD machine (the MEIKO CS-1) and on a SIMD machine (the MASPAR MP-1).

As regards the MIMD machine, the Transputer array programmed in OCCAM2 has been configured as a ring and a network with 256 input neurons, 128 hidden

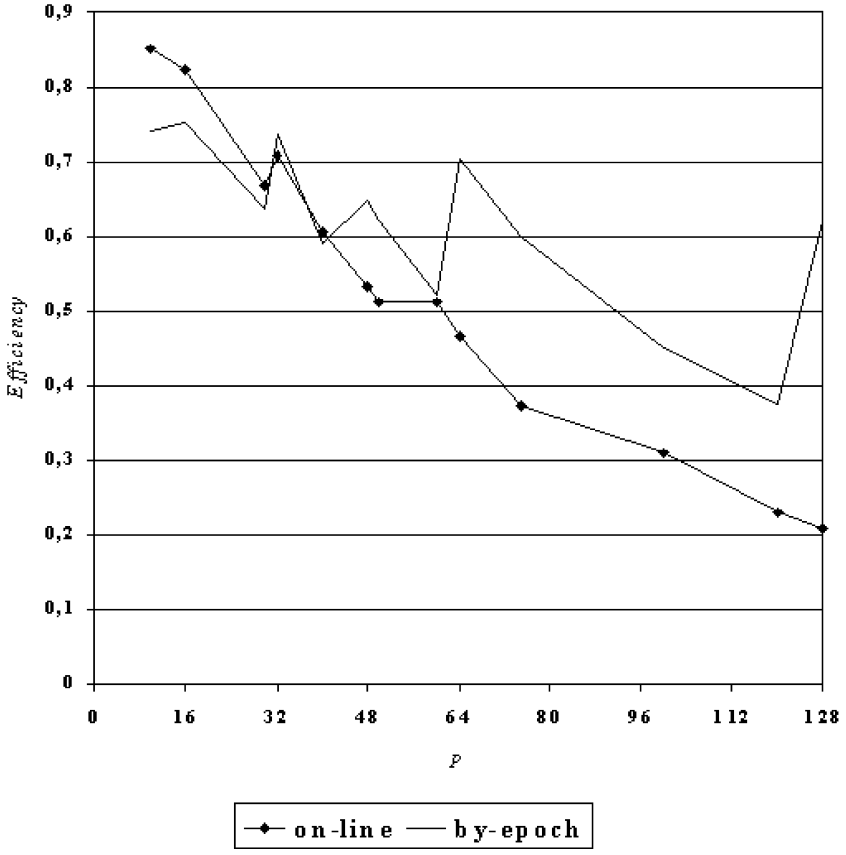

Fig. 7. The efficiency obtained on a ring connected Transputer array ($Ni = 256, Nh = 128, No = 16$).

neurons and 16 output neurons has been chosen as a case-study; moreover, it has been supposed that the training set is composed by 100 examples (each consisting of 256 single precision floating point values) and the algorithm has been tested by performing 100 iterations. The results obtained are shown in Fig. 7, where the efficiency of the on-line version of the BPA and of the by-epoch version are compared. The local maxima are due to the fact that the computational complexity of the parallel algorithm is a function of $\lceil Nh/P \rceil$. In the by-epoch version the forward–backward parallelism has been used; thus, each process iteratively performs the following steps (this solution is referred in the following as *sol*_1):

- parallel on different patterns:

1. evaluates the activation value of the handled hidden neurons;
2. evaluates the partial sums for the calculation of the net input to output neurons;
3. sends the evaluated vector of partial sums to the master;

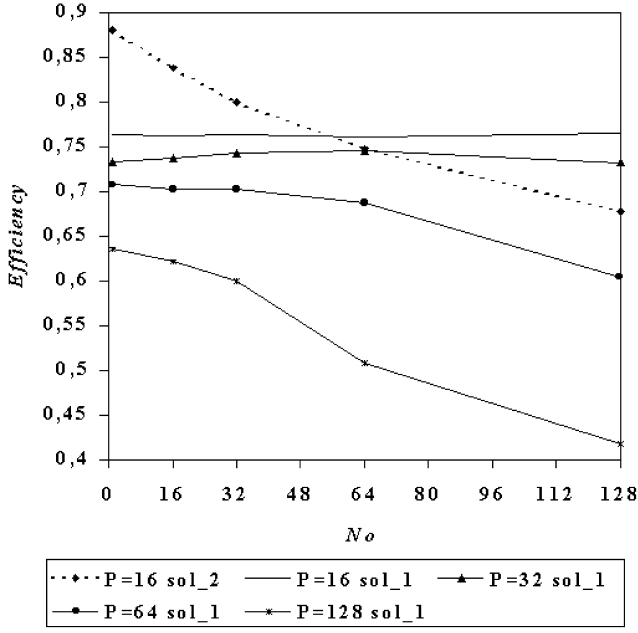

Fig. 8. The efficiency as a function of No on a ring-connected Transputer array ($Ni = 256, Nh = 128$).

- and

1. waits for the arrival of error terms of output neurons;
2. evaluates the error term of the handled hidden neurons;
3. evaluates and accumulate weight changes.

The handling of the parallel among processes clearly introduces an overhead; in the case under study, in fact, if P is lesser than about 32, the efficiency is greater in the on-line version. Obviously, this means that, if the learning to be used is by-epoch and if P is less than 32, it is better to use the on-line version that can be easily modified to simulate the batch version (this implementation is referred in the following as *sol*_2). The efficiency obtained using the *sol*_2 is slightly better than that obtained in the on-line case (for example, if $P = 16$ we obtain an efficiency of 0.823 in the on-line case and an efficiency of 0.838 in the *sol*_2 case); this is due to the fact that the overhead due to the sum of weight changes to weights is completely parallelized. Fig. 8 shows the efficiency for the test-case ($Ni = 256$, $Nh = 128$) as No varies for by-epoch learning; as regards the *sol*_1, the efficiency of course decreases as No increases since, even if communication and elaboration are overlapped, the overhead due both to the communication and to the evaluation of error terms of output neurons linearly increases with No. However, if Nh/P is sufficiently large (at least equal to 4 in our test-case), such an effect is not perceptible. As regards the case $P = 16$ the efficiency increases if the *sol*_2 is adopted, as it has been described above. In this case, on the other hand, the efficiency decreases as No increases since the processors are fully idle

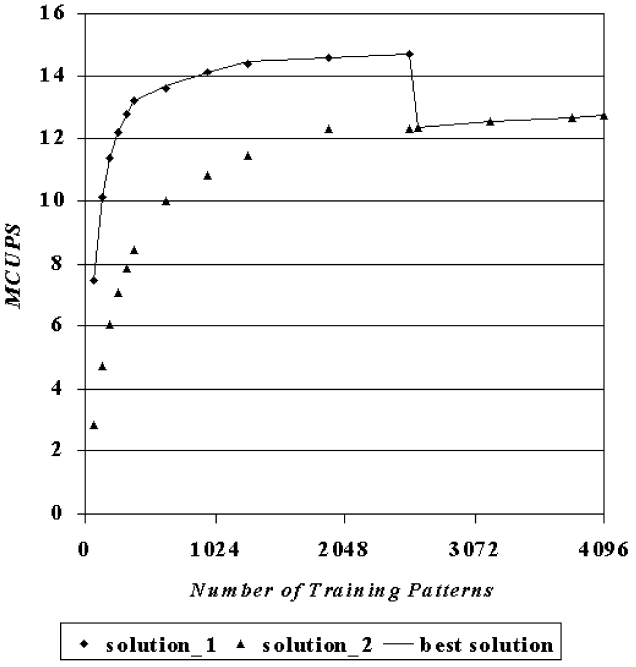

Fig. 9. The MCUPS as a function of the number of training patterns obtained on the MASPAR MP-1 ($Nh = 256, Nh = 128, No = 16$, learning by-epoch).

during communication; if No is at least equal to 64 *sol*_1 re-becomes the best solution. By the way, it has to be noted that in Fig. 8 there are no local maxima; this is simply due to the fact that the algorithms have been tested for values of P such that Nh/P is integer.

The MASPAR MP-1 is composed (in our configuration) by 4096 mesh-connected 4-bit processors, each of which has a local memory of 16 Kbytes; the mesh is square (64 rows and 64 columns) and the machine has a peak performance (in our configuration) of 6400 MIPS and 300 MFLOPS (both on 32 bit). Using this machine, clearly, the hypotheses in Section 4 do not hold; in fact the memory of each processor cannot store the whole training set and the number of processors available is greater than Nh. Because of this, it has been supposed (as regards again to a network with 256 input neurons, 128 hidden neurons and 16 output neurons) that the learning is by-epoch; moreover it has been supposed that each training pattern consist of 256 one-byte (unsigned) integer numbers. On each row of the mesh a replica of the network has been mapped and two implementations have been developed. In the first one (*solution*_1) it has been supposed that all training patterns are stored in the memory of each processor; in this case at most 40 examples can be stored and thus at most $64 * 40 = 2360$ training patterns can be used. In the second solution (*solution*_2) the training set is divided into $\beta = \alpha/4096$ sub-sets (α represents the number of training patterns) and in the memory of each processor are stored β patterns. The MCUPS

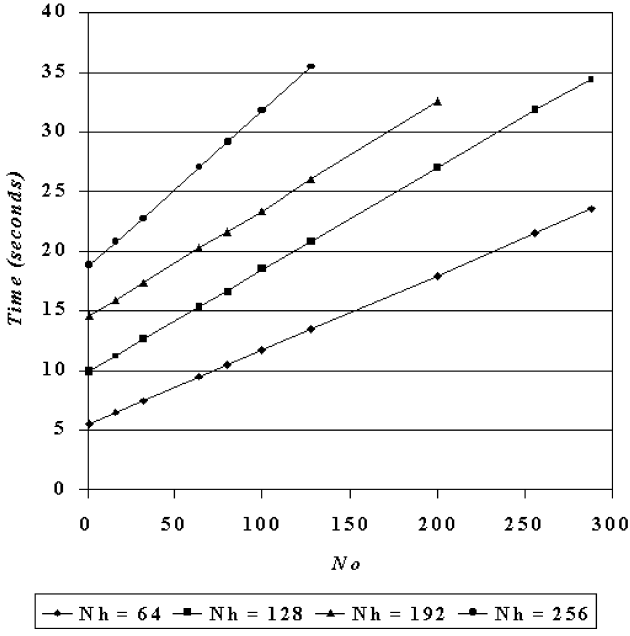

Fig. 10. The performance obtained by using the MASPAR MP-1 as *No* vary ($Nh = 256$, one by-epoch iteration on 4096 training patterns).

obtained using the two solutions are compared in Fig. 9; clearly, *solution_1* is faster but *solution_2* allows to deal with more training patterns. It is worth noting that the performance increases as the number of examples increases; this effect is by itself not surprising since, as the number of training patterns increases, the overhead due to the sum of weight changes to weight becomes more and more negligible. It is however a matter of fact that the effect is here amplified; this is a consequence of the fact that, for the combination of weight changes, it is not possible to use *scan* facilities.

Last, Fig. 10 shows (with reference to a network with 256 input neuron, 128 hidden neuron and 4096 training patterns) the time as a function of *No*; such a function is of course a linear function of *No*.

6. Concluding remarks

In this paper a new mapping scheme has been proposed for the parallel implementation of the learning phase of feed-forward neural networks. This mapping scheme (Clepsydra) allows to develop implementations oriented both to MIMD and SIMD parallel computers; simulation results on a Transputer-based MIMD machine and on a SIMD machine (the MASPAR MP-1) have been also sketched and commented.

The Clepsydra mapping scheme is based on the use of all-to-one (vectorial) associative broadcasting and on the use of one-to-all (vectorial) broadcasting. Since primitives for such kind of communication are widely available and are very powerful, the proposed scheme allows to fully exploit the computing power available; for example, it has been shown that, even with reference to a toy-example, a ring-connected array of 64 T800 Transputers can be used with an efficiency of about 0.75.

An advantage offered by our scheme, perhaps not well addressed in the paper, is that the *parallel* code could be written as easily as the *sequential* code; what is required to have the parallel simulator simply is to modify the forward phase of the network you are dealing with, provided obviously that the algorithm can set the number of the hidden neurons at run-time.

In the opinion of the author, to fully exploit the potentialities of the Clepsydra mapping scheme it is necessary to design and realize a dedicated machine. Such a special-purpose neurocomputer should be composed by commodity processors and by two communication networks. The first communication network must be able to perform the combination of data and it has to work in parallel with the processors; in other words, once the first component of the vectors $\mathbf{J}^i$ has been evaluated, the combination of these data must be performed without interacting with the computation of the second component of the vectors $\mathbf{J}^i$ (and so on). The other communication network must be used for the broadcasting of error terms to processors; in this case it is sufficient to have a global bus, whose arbitration is very simple, since only the master processor can write on it. Last, each processor must have a local memory in which connection weights are stored and, as regards the storing of training patterns, we could suppose that a large shared memory should be provided; again the arbitration of such a memory is not a real problem since, once training patterns have been loaded in it at start-up time, such a memory becomes a read-only memory.

The main drawback of the proposed scheme (when it is used with general-purpose computers) lies in the fact that each processor has to know the whole training set. Even if such an hypothesis can be often satisfied through a careful organization of learning data, it is a matter of fact that in several real cases it must be removed and this clearly diminishes both the performance and the easiness of coding. Another actual problem clearly concerns the use of parallel machines with $P > Nh$; here, in fact, it is not yet possible to eliminate the need of batch learning. To actually remove these constraints, the author is developing a generalized version of the Clepsydra scheme and this is the main concern of his work in progress.

References

[1] E. Ahmed, K. Priyalal, Algorithmic mapping of feedforward neural networks onto multiple bus systems, IEEE Trans. Parallel Distrib. Systems 8 (2) (1997) 130–136.

[2] M. Besch, H.W. Pohl, Flexible data parallel training of neural networks using MIMD-computers, in Third Euromicro Workshop on Parallel and Distributed Processing, Sanremo, Italy, January 1995.

[3] J. Bourrely, Parallelization of a neural learning algorithm on a hypercube, in: F. Andre, J.P. Verjus, (Eds.), Hypercube and Distributed Computers, Elsevier Science Publishers B. V., North-Holland, 1989, pp. 219–229.

[4] U. De Carlini, U. Villano, Transputers and Parallel Architectures, Ellis Horwood, Chichester, England, 1991.
[5] G. Fox, S. Otto, A. Hey, Matrix algorithm on a hypercube: matrix multiplication, Parallel Comput. 4 (1987) 17–31.
[6] Y. Fujimoto, N. Fukuda, T. Akabane, Massively parallel architectures for large scale neural networks simulations, IEEE Trans. Neural Networks 3 (6) (1992) 876–888.
[7] E.J.H. Kerckhoffs, F.W. Wedman, E.E.E. Frietman, Speeding up back-propagation training on a hypercube computer, Neurocomputing 4 (1992) 43–63.
[8] H. Klapuri, T. Hamalainen, J. Saarinen, J. Kaski, Mapping artificial neural networks to a tree shape neurocomputer, Microprocessors Microsystems 20 (5) (1996) 267–276.
[9] V. Kumar, S. Shekhar, M.B. Amin, A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures, IEEE Trans. Parallel Distrib. Systems 5 (10) (1994) 1073–1090.
[10] S.Y. Kung, J.N. Hwang, Parallel architectures for artificial neural nets, Proceedings of the ICNN, San Diego, CA, 1988, vol. 2, pp. 165–172.
[11] S.Y. Kung, J.N. Hwang, A unified systolic architecture for artificial neural networks, J. Parallel Distrib. Comput. 6 (1989) 358–387.
[12] T. Nordstrom, B. Svensson, Using and designing massively parallel computers for artificial neural networks, J. Parallel Distrib. Comput. 14 (3) (1992) 260–285.
[13] A. Petrowski, L. Personnaz, L., G. Dreyfus, C. Girault, Parallel implementations of neural network simulations, Hypercube and Distributed Computers, Elsevier Science Publishers B. V., North-Holland, 1989, pp. 205–218.
[14] D.A. Pomerleau, G.L. Gusciora, D.S. Touretzky, H.T. Kung, Neural network simulation at warp speed: how we got 17 million connections per second, Proceedings of the ICNN, San Diego, CA, 1988, vol. 2, pp. 581–586.
[15] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representation by back-propagation of errors, Nature 323 (1986) 533–536.
[16] D.E. Rumelhart, J.L. McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, MIT Press, Cambridge, MA, 1986.
[17] A. Singer, Implementations of artificial neural networks on the connection machine, Parallel Comput. 14 (1990) 305–315.
[18] M. Witbrock, M. Zagha, An implementation of back-propagation learning on GF11, a large SIMD parallel computer, Parallel Comput. 14 (1990) 329–346.
[19] X. Zhang, M. Mckenna, J.P. Mesirov, D.L. Waltz, The back-propagation algorithm on grid and hypercube architectures, Parallel Comput. 14 (1990) 317–327.

Antonio d'Acierno received the "laurea" degree *cum laude* in Electronic Engineering at the University of Naples (Italy) in 1988. Since then he has been with the "Istituto per la Ricerca sui Sistemi Informatici Paralleli" of the Italian National Research Council. His research activity concerns parallel architectures for neural networks simulation, parallel architectures for image processing, distributed object oriented programming and object oriented data base management systems. Antonio d'Acierno is a member of IEEE Computer Society and of ACM.