Mapping Neural Network Back-Propagation onto Parallel Computers with Computation/Communication Overlapping

B. Girau

Laboratoire d'Informatique du Parallélisme - CNRS URA 1398
46 allée d'Italie, 69364 Lyon cedex 07, France
phone: +33 72 72 85 47 fax: +33 72 72 80 80 email: bgirau@lip.ens-lyon.fr
URL: http://www.ens-lyon.fr/~bgirau

Abstract. It is shown in [6] that mapping neural networks onto existing parallel computers leads to an unsatisfactory efficiency, except for irregularly connected networks, with several distinguishable highly connected regions. This paper shows how a four step decomposition of the back-propagation algorithm allows to introduce computation/communication overlapping so as to improve any parallel mapping of differentiable feedforward neural networks. This solution can adapt to both irregular and regular feedforward neural structures. Its computational complexity is estimated for multilayered networks. Unlike most network partitioning schemes, it can deal with multilayer networks using non-standard neurons, such as wavelet networks. Unlike a pattern partitioning algorithm, it is able to implement the stochastic gradient learning algorithm. Numerical results show that this solution should be considered as soon as communication overlapping is available.

1 Introduction

Many algorithms have been developed so as to parallelize neural network applications. A satisfactory survey of existing schemes to parallelize back-propagation can be found in [11]. The choice of the best parallelization method strongly depends on both neural network application and employed parallel machine. See [14] for a comparison between some standard methods.

Pattern partitioning schemes are coarse-grained methods. They require large pattern sets, and they are not able to implement a *stochastic gradient learning* algorithm, where the neural network parameters are updated after each pattern presentation. See [13] for a study on a ring, [9] for an improved version, and [7] for a study on several different parallel architectures.

Network partitioning schemes are medium to fine-grained methods. They apply to large neural networks. Their efficiency frequently depends on the density of the neural network connections. Some of them use efficient parallel implementations of the algebraic computations that are performed in a multilayer perceptron (MLP). See for instance [15], or [11, 18] in a less obvious way. Other ones try to *map* the natural parallelism of a neural network onto the parallel

computer, by partitioning the neurons among the processors. A general study of this solution has been performed in [6]. It leads to a rather pessimistic conclusion about the efficiency of a neural network *mapping* onto existing machines. Moreover, such a mapping is ill-adapted to regular neural network structures such as MLPs, which are among the most employed neural networks in standard applications. A rather intuitive mapping for MLP is proposed in [16], but its satisfactory results are obtained on a particular parallel computer and it uses beyond measure MLPs.

The aim of this paper is to show how a precise study of the back-propagation algorithm allows an efficient *mapping*. It takes advantage of a general form of the back-propagation so as to introduce computation/communication overlapping. It applies to a general feedforward neural network model, including MLP, wavelet networks, RBF networks, and any irregularly connected feedforward neural network. A processing time model is provided for regular multilayer structures. The main drawback appears in the minimum size of the neural network layers that is required to allow the computation/communication overlapping. Numerical results show that, thanks to the introduced communication overlapping, the general unsatisfactory efficiency of neural network mappings is overcome for the particular case of the back-propagation implementation of large enough neural networks.

Section 2 shortly describes the general neural network model and the associated back-propagation. Section 3 focuses on the new parallelization method. Section 4 deals with the case of regular multilayer neural networks.

2 General feedforward neural networks

2.1 Neural network learning

In a learning phase, a pattern set is given. It contains inputs and the corresponding expected outputs. A learning iteration starts with an error computation. It estimates the difference between the expected outputs and the computed outputs for some selected patterns. Then the network parameters are modified to reduce this error. Several learning iterations are performed, with the same patterns or with different ones.

The error function is assumed to be differentiable. Then a *gradient descent* algorithm can be used for the learning iterations. If $P(t)$ is the parameter vector of the network at time t, and if $\mathcal{GP}(t)$ is the gradient of the error function with respect to the coordinates of $P(t)$, then $P(t+1) = P(t) - \epsilon \mathcal{GP}(t)$, where $\epsilon \in \mathbb{R}_+^*$. If the error function is computed for only one pattern, then the *stochastic gradient* algorithm is used (a new pattern is randomly chosen for each learning iteration). If the whole pattern set is considered, then it is called the *total gradient* algorithm. If every learning iteration uses only part of the pattern set, and if the size of this part is constant, then a *block-gradient* algorithm is used. It has often been pointed out that the convergence time of a neural network learning grows with the number of patterns handled by the error computation. Therefore, the fastest learning is the stochastic one.

The back-propagation algorithm is often employed, since many neural network applications use MLPs with a gradient descent learning and a *back-propagated* computation of the gradient.

2.2 Feedforward neural networks

A theoretical study of a general feedforward neural network model is proposed in [4, 5]. It shows that the back-propagation algorithm can adapt to any differentiable feedforward neural network. A MLP is only a particular feedforward neural network.

In this theory, a feedforward neural network is a DAG (directed acyclic graph), in which the predecessor set of each node has been totally ordered. The vertices are differentiable vectorial functions $f_i(x_i, p_i)$, $1 \leq i \leq I$, where x_i is the input vector of vertex i, and p_i is its parameter vector. Layers can be defined so that any f_i in layer l can compute its output as soon as all f_j in layers $0, \ldots, l-1$ have computed their own outputs. If f_i has no predecessor in the graph, then it is an input vertex, and it belongs to the first layer. Else its input is obtained by appending together the outputs of all its predecessors. Then the whole neural network $\mathcal{N}$ is defined as follows: the inputs of the input vertices form the input vector of $\mathcal{N}$, the parameter vector of $\mathcal{N}$ is $P(t) = (p_1, \ldots, p_I)$, and the output of $\mathcal{N}$ is obtained by appending together the outputs computed by the vertices that have no successor in the graph. To simplify, functions f_i may be called *neurons*. This formal definition allows to show that the function computed by a neural network may be considered as a neuron.

2.3 Back-propagation

In any general feedforward neural network, $\mathcal{GP}(t)$ can be computed thanks to a back-propagated algorithm, which is faster than the natural gradient computation that considers $\mathcal{N}$ as a composite function. See [4] for a mathematical description of this general back-propagation, and a full study of its complexity. This algorithm requires *local* operations (it may be described as a sequence of computations performed by each neuron thanks to local data only).

Let $\mathcal{GP}_i$ be the gradient of the error function with respect to p_i. Let $\mathcal{GX}_i$ be the gradient of the error function with respect to x_i. Let $f_1^{(i)}, \ldots, f_{s_i}^{(i)}$ be the successors of f_i in the graph. For each $j \in \{1, \ldots, s_i\}$, let $\mathcal{GX}_j^{(i)}$ be the gradient of the error function with respect to the input of f_j that is received from f_i (i.e., $\mathcal{GX}_j^{(i)}$ is the part of $\mathcal{GX}_j$ that corresponds to the connection from f_i to f_j). Let $\mathcal{JX}_i$ be the matrix of the local differential (jacobian matrix) of f_i with respect to its input and let $\mathcal{JP}_i$ be the local jacobian matrix of f_i with respect to its parameter. Then the back-propagation theorem implies that:

$$\mathcal{GP}_i = \sum_{j=1}^{s_i} \mathcal{GX}_j^{(i)} \mathcal{JP}_i \quad \text{and} \quad \mathcal{GX}_i = \sum_{j=1}^{s_i} \mathcal{GX}_j^{(i)} \mathcal{JX}_i$$

According to this mathematical description, two steps can be defined for the local computation of each neuron:

1. Within the forward step, a neuron waits for its inputs from its predecessors, then it computes its own output and forwards it to its successors.
2. Within the backward step, it waits for the gradients from its successors. Then it applies their sum to its local differentials, and thus obtains the gradient of the error function with respect to its parameters and to its inputs. It sends the latter ($\mathcal{G}X_i$) backwards to its predecessors.

3 Back-propagation and mapping

3.1 Computation/communication overlapping

Let $\mathcal{N} = \{f_1, \ldots, f_I\}$ be a feedforward neural network. Let $\mathcal{M}$ be a mapping of $\mathcal{N}$ onto a parallel computer: $\mathcal{M}(f_i)$ is the processor on which neuron f_i has been mapped.

Principle

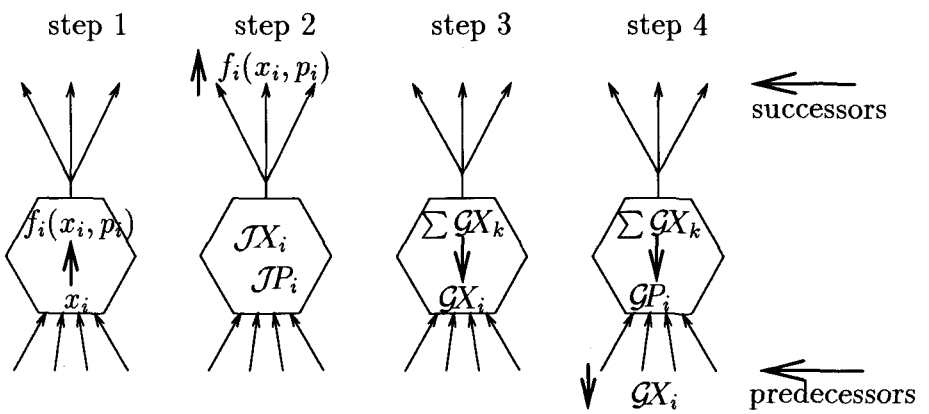

Fig. 1. Four steps of back-propagation

Instead of performing only two steps, forward and backward, each **local** computation can be divided into four steps (see figure 1):

1. Neuron F_i computes its output ($f_i(x_i, p_i)$) thanks to the received inputs (x_i) and then it sends this output to its successors.
2. Neuron F_i computes its local differentials, $\mathcal{J}X_i$ and $\mathcal{J}P_i$, which only depend on the forwarded data (i.e., the already received inputs).

3. Thanks to the backpropagated gradient ($\sum \mathcal{G}X_k$) that have been sent by its successors, neuron F_i computes the gradient of the error function with respect to its input ($\mathcal{G}X_i$), and then it sends this gradient to its predecessors.
4. Neuron F_i computes the gradient of the error function with respect to its parameter vector ($\mathcal{G}P_i$).

This decomposition allows an interesting computation/communication overlapping, if the machine allows asynchronous non-blocking communications:

> Each neuron may send its computed output while it computes its local differentials (the computation of step 2 overlaps the forward communication). In the same way, it may send the gradient of the error function with respect to its inputs while it computes the gradient of the error function with respect to its parameters (the computation of step 4 overlaps the backward communication). Moreover, the neural network can be locally updated within step 4 by means of $p_i(t+1) = p_i(t) - \epsilon\, \mathcal{G}P_i(t)$.

Algorithm

Each processor p performs the following algorithm for one iteration of the back-propagation. *This algorithm will be called* CCO *algorithm from now on.*

FIRST INITIALIZATION: I_p is the set of input neurons that $\mathcal{M}$ maps on p.
FIRST LOOP:
do: 1. Step 1 performed for each neuron in I_p
2. Simultaneously:
- Step 2 performed for each neuron in I_p.
- Non-blocking one-to-all (worst case) communication to communicate the computed outputs to the other processors. Indeed, this message has to be received only by the processors that contain successors of any neuron in I_p.

3. I_p updated as follows: it is the set of the neurons mapped by $\mathcal{M}$ on p, for which steps 1 and 2 have not been performed and for which all required inputs have been received from the other processors.

until: Steps 1 and 2 have been performed for every neuron mapped on p.
SECOND INITIALIZATION: I_p is the set of output neurons mapped by $\mathcal{M}$ on p.
SECOND LOOP:
do: 1. Step 3 performed for each neuron in I_p
2. Simultaneously:
- Step 4 performed for each neuron in I_p (possibly including the local parameter updating).
- Non-blocking one-to-all (worst case) personal communication to communicate the computed gradients to every other processor.

3. I_p updated as follows: it is the set of the neurons mapped by $\mathcal{M}$ on p, for which steps 3 and 4 have not been performed and for which all required back-propagated gradients have been received from the other processors.

until: Steps 3 and 4 have been performed for every neuron mapped on p.

In this algorithm, several patterns may be simultaneously handled. In this case, each communication gathers the data for all patterns.

Implementations of gradient descent algorithms

Any gradient descent algorithm can be implemented with a neural network parallel mapping, and therefore with the CCO algorithm. Each computation step may be processed for a single pattern as well as for the whole pattern set, provided that it does not exceed the per-processor memory. Another way to introduce computation/communication overlapping is to pipeline the patterns, but this solution can not implement the stochastic gradient algorithm.

3.2 Suitable mappings and parallel architectures

The efficiency of the algorithm depends on the neural network and on the chosen mapping $\mathcal{M}$. The choice of $\mathcal{M}$ chiefly depends on the neural network structure, but it depends on the parallel computer architecture too.

Architecture

A good criterion to choose the parallel architecture is the total number of messages implied by both all-to-all communication (multinode broadcast within step 2) and all-to-all personal communication (multinode scattering within step 4). If a non-blocking communication call is modelled as a constant startup time, and if the transfer time is overlapped by computation, then the communication cost of the CCO algorithm is proportional to the number of messages. Table 1 recalls both number of messages and communication times for some standard architectures. The hypercube structure should be chosen if available. Indeed, hypercube-like communications can be simulated on other hardware architectures without loss of performance (e.g. on the Cray T3D).

		ring	grid/torus	hypercube
all-to-all	messages	$\mathcal{O}(p)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(\log(p))$
	total time	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$
personal all-to-all	messages	$\mathcal{O}(p)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(\log(p))$
	total time	$\mathcal{O}(p^2)$	$\mathcal{O}(p\sqrt{p})$	$\mathcal{O}(p\log(p))$

Table 1. Communication costs for several parallel architectures

Since each communication uses several messages, the computation of steps 2 and 4 has to be sliced, so that the parallel implementation performs the computational operations and the non-blocking communication calls by turns. Reliable computation and communication time models are required to obtain a precise overlapping.

Mapping

Any feedforward neural network implies sequential computation (e.g. neurons in layer l have to wait for neurons in layer $l-1$ in the forward step). Therefore, a good mapping probably performs a quite vertical sectioning, i.e., it partitions the neurons of each layer quite regularly among the processors.

For very irregular structures, the study of [6] provides suitable mappings. Another method is to consider any feedforward neural network as a reversible task graph, and then to use any standard scheduling algorithm. See [7] for a study of the Modified Critical Path scheduling with both forward and backward phases.

For regular structures, such as multilayer neural networks, the strict vertical sectioning (see next section) is a satisfactory mapping. In the *checkerboarding* method proposed by [11], each neuron is mapped onto several processors. But this solution limits itself to multilayer perceptrons, since they perform large matrix-vector products (that do not appear in a wavelet network for instance).

4 Multilayer networks

4.1 Notations

Let L be the number of layers. Let n_l be the number of neurons in layer l. Each neuron in layer 1 receives n_0 inputs, which are the network inputs. For each $l \in \{2, \ldots, L\}$, each neuron in layer l receives n_{l-1} inputs, and computes one output in $\mathbb{R}$. These inputs are the outputs of the neurons in layer $l-1$. Therefore, consecutive layers are fully connected. The outputs of the neurons in layer L are the outputs of the network. The standard quadratic error function is used. All neurons are identical (f_i does not depend on i).

It is assumed that the computation time of a neuron is linear with respect to its number of parameters. In the same way, the computation time of its differentials is linear with respect to its number of parameters. It is also assumed that its number of parameters is proportional to its input size. Sigmoid neurons, wavelet neurons and RBF neurons satisfy these conditions.

Therefore, the computation time of step j ($1 \leq j \leq 4$) will be modelled as: $\gamma_j + \delta_j n_{l-1}$ for any neuron in layer l (updating might be taken into account by step 4). For the last layer, the differential of the error function has to be computed: δ_e for one output neuron.

A *vertical sectioning* with p processors is used. Each processor deals with n_l/p neurons in layer l. To simplify, it is assumed that each n_l is a multiple of p. In practice, if a layer contains too few neurons, it may be interesting to map the whole layer onto one processor, and to perform only one-to-all and all-to-one communications for this layer. Indeed, standard applications often have small numbers of output neurons.

In what follows, the diameter of the parallel architecture is called d. It is equal to $\log_2(p)$ for a hypercube, and $2\sqrt{p}$ for a grid. A constant time model, $\beta_{c/c}$ is used for the non-blocking communication calls. With existing machines,

this model is simple, but satisfactory. See [7] for a full study with the Intel iPSC 860 (also available in my URL).

4.2 Computational complexity

Let k be the number of simultaneously handled patterns.

STEP 1 FOR LAYER l:	$k(\gamma_1 + \delta_1 n_{l-1})\frac{n_l}{p}$
STEP 2 FOR LAYER l $(l < L)$:	$k(\gamma_2 + \delta_2 n_{l-1})\frac{n_l}{p} + 2d\beta_{c/c}$ (no transfer time, on account of overlapping)
STEP 2 FOR LAYER L:	$k(\gamma_2 + \delta_2 n_{L-1})\frac{n_L}{p}$
STEP 3 FOR LAYER L:	$k(\gamma_3 + \delta_3 n_{L-1} + \delta_e)\frac{n_L}{p}$
STEP 3 FOR LAYER l $(1 < l < L)$:	$k(\gamma_3 + \delta_3 n_{l-1})\frac{n_l}{p}$
STEP 3 FOR LAYER 1:	0, since the gradient with respect to the network inputs is useless.
STEP 4 FOR LAYER l $(1 < l)$:	$k(\gamma_4 + \delta_4 n_{l-1})\frac{n_l}{p} + 2d\beta_{c/c}$
STEP 4 FOR LAYER 1:	$k(\gamma_4 + \delta_4 n_0)\frac{n_1}{p}$

The expected speedup is then $\dfrac{p\mathrm{T}_{\mathrm{seq}}}{\mathrm{T}_{\mathrm{seq}} + 2(L-1)dp\beta_{c/c}}$, with

$$\mathrm{T}_{\mathrm{seq}} = k\left[\Gamma(\sum_{l=2}^{L} n_l) + \Delta(\sum_{l=2}^{L} n_{l-1}n_l) + \delta_e n_L + (\Gamma - \gamma_3)n_1 + (\Delta - \delta_3)n_0 n_1\right]$$

where $\Gamma = \sum_{i=1}^{4} \gamma_i$ and $\Delta = \sum_{i=1}^{4} \delta_i$.

4.3 Minimum layer size

The first condition to obtain this speedup is that the communication overlapping is allowed. Therefore, the computation loads have to exceed the message transfer times. If τ is the transfer time for one real,two conditions must be satisfied:

$$\text{overlapping within step 2:} \quad \forall l \in [1, L-1] \quad 2p\tau \leq (\gamma_2 + \delta_2 n_{l-1}) \qquad (1)$$

$$\text{overlapping within step 4:} \quad \forall l \in [2, L] \quad 2dpn_{l-1}\tau \leq (\gamma_4 + \delta_4 n_{l-1})n_l \qquad (2)$$

For the particular case of a MLP, an optimization may be introduced. Step 2 may be enclosed in both step 3 and step 4 without significant loss of time, since these steps are equivalent to scalar multiplication applied to both local input and parameter vectors. Therefore, step 2 only performs an all-to-all communication, whereas step 4 communication is still overlapped by computation. Dealing with a MLP is *the worst case for the* CCO *algorithm.*

4.4 Results

Comparison with network-partitioning schemes

Various works were considered, such as [16] (based on the same vertical partitioning as in section 4, but outperformed by [11]), [14], [15] or [18]. The checkerboarding method of [11] (hereafter called CB algorithm) is finally taken as a reference, with regard to its high efficiency and scalability. Of course, only the case of a MLP is considered, since the CB method only applies to this type of neural network. Therefore, only condition 2 is taken into account to estimate the minimum layer size, since communications are not overlapped in step 2. The minimum values of the number of neurons are reported on figure 2.

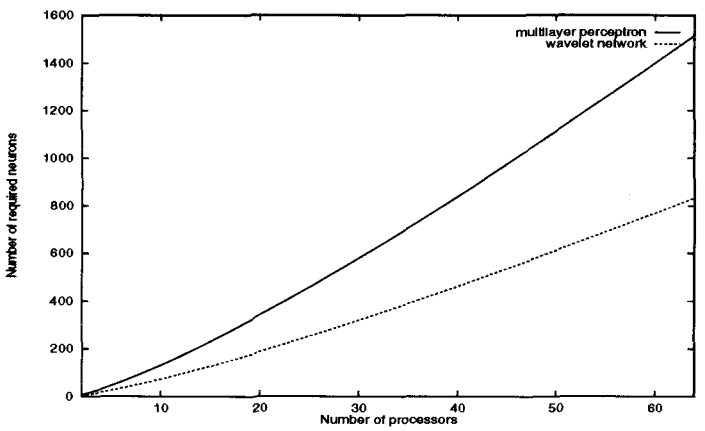

Fig. 2. Minimum number of neurons in each layer for the CCO algorithm

Comparison conditions

Speedups are computed with the same number n of neurons in each layer of the MLP. For this paper, experiments have been performed on an iPSC 860. The theoretical performance models of this paper and of [11] are therefore applied to this particular machine.

Speedups are given for 2-layer perceptrons in tables 2, 3 and 4. The box remains empty when overlapping conditions are not fulfilled. For the experimental results of table 4, the speedups are computed with respect to the theoretical sequential processing time, when the problem size does not allow one processor to handle as large MLPs as the parallel implementation can do. Moreover, the box remains empty when the neural network requires too much per-processor memory even for the parallel implementation.

In many experiments for the CCO algorithm, the per-processor memory limit did not allow to perform the computation for several patterns within a single

forward-backward processing (too many data to store for each layer). In this case, the different patterns had to be processed one after each other. Therefore, all the given speedups of both CCO and CB methods are given for $k = 1$. It corresponds to the worst case for the theoretical speedup of both algorithms, but it shows what happens when the stochastic gradient is implemented.

	$p = 4$	$p = 16$	$p = 64$
$n = 30$	1.68	1.51	1.17
$n = 250$	3.78	12.39	28.45
$n = 1500$	3.97	15.51	57.04

Table 2. MLP: theoretical speedups for the checkerboarding parallelization ($k = 1$)

	$p = 4$	$p = 16$	$p = 64$
$n = 30$	2.26		
$n = 250$	3.85	13.1	
$n = 1500$	3.98	15.57	57.13

Table 3. MLP: theoretical speedups with the CCO algorithm ($k = 1$)

	$p = 4$	$p = 16$	$p = 64$
$n = 30$	1.19		
$n = 250$	3.65	13.3	
$n = 1500$		15.51	56.55

Table 4. MLP: experimental speedups with the CCO algorithm ($k = 1$)

	$p = 4$	$p = 16$	$p = 64$
$n = 30$	3.81		
$n = 250$	3.99	15.91	
$n = 1500$	4	16	63.94

Table 5. WN: theoretical speedups with the CCO algorithm ($k = 1$)

	$p = 4$	$p = 16$	$p = 64$
$n = 30$	3.51		
$n = 250$	3.93	15.68	
$n = 1500$		15.94	63.55

Table 6. WN: experimental speedups with the CCO algorithm ($k = 1$)

	$p = 4$	$p = 16$	$p = 64$
$k = p * 1$	0.44	0.39	0.38
$k = p * 10$	2.20	3.21	3.65
$k = p * 100$	3.69	11.45	24.13

Table 7. MLP: theoretical speedups with pattern-partition ($n = 250$)

	$p = 4$	$p = 16$	$p = 64$
$k = p * 1$	1.60	1.93	2.04
$k = p * 10$	3.48	9.25	15.88
$k = p * 100$	3.94	14.91	49.12

Table 8. WN: theoretical speedups with pattern-partition ($n = 250$)

Result comment

Overlapping provides very satisfactory performance, slightly better than the CB algorithm. But the CCO solution is available for a minimum number of neurons which is proportional to $p \log(p)$, whereas CB only requires a number of neurons proportional to $\sqrt{(p)}$.

Some loss of performance appears with experiments, when compared with

the theoretical performance model. It can be explained by the fact that there is no autonomous DMA in an iPSC 860. Therefore, non-blocking communications imply a limited loss of performance during overlapping computation.

Comparison with pattern-partitioning schemes

Results for MLP are given for $n = 250$ in table 7 and in the second row of table 3. Results for wavelet networks (WN) are given for $n = 250$ in table 8 and in the second row of table 5. Speedup formulae for pattern-partitioning on hypercube architectures are taken from [7]. Let us recall that with the pattern partition, if each processor deals with b patterns, then the number of handled patterns is $k = p * b$.

Result comment

The performance of pattern-partitioning collapses when small blocks of patterns are handled and when many processors are used. Wavelet networks imply so much computation[1] that the parallel efficiency is usually greater than 99% when the CCO algorithm is used. Communications are easily overlapped by computations for this particular neural network.

When available, the CCO algorithm outperforms the pattern-partitioning method, which should still be used with reduced-sized neural networks and large training sets.

5 Conclusion

This paper introduces computation/communication overlapping in any parallel mapping of the neural network back-propagation algorithm. Overlapping appears thanks to a 4-step decomposition of this algorithm, whereas standard decompositions only use two steps.

It allows some performance improvement for MLP when it is compared to other network-partitioning schemes. But its main drawback is to require a number of neurons proportional to $p \log(p)$, if p is the number of useful processors. Nevertheless, unlike most network-partitioning schemes (particularly unlike the most efficient ones), it can be applied to any feedforward neural network.

The pattern-partitioning scheme also applies to any neural network structure, but its efficiency collapses when a small number of patterns is used. The parallel solution of this paper allows to use such small pattern sets. Moreover, it can implement the stochastic gradient, so that the learning time is reduced.

Some modified versions of this CCO implementation are under development. They reduce the number of required neurons per layer.

[1] for neuron i in layer l (see [17] for an introduction to wavelet networks), $\prod_{j=1}^{n_{l-1}} h(d_j^{(i)}(x_j - \theta_j^{(i)})$, where $h(x) = -x \exp(-x^2/2)$. The $d_j^{(i)}$ and the $\theta_j^{(i)}$ are the parameters.

References

1. L. Bottou and P. Gallinari. A Framework for the Cooperation of Learning Algorithms. In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Neural Information Processing Systems*, volume 3, pages 781–788. Morgan Kauffman, 1991.
2. M. Cosnard, J.C. Mignot, and H. Paugam-Moisy. Implementations of multilayer neural networks on parallel architectures. In *Proc. 2^{nd} IEE Int. Spec. Seminar on the Design and Application of Parallel Digital Processors*, pages 43–47, April 1991.
3. C. Gégout, B. Girau, and F. Rossi. NSK, an object-oriented simulator kernel for arbitrary feedforward neural networks. In *ICTAI Int. Conf. on Tools with Artificial Intelligence*, pages 95–104. IEEE Computer Society Press, 1994.
4. C. Gégout, B. Girau, and F. Rossi. A general feedforward neural network model. Technical report NC-TR-95-041, NeuroCOLT, Royal Holloway, University of London, 1995.
5. C. Gégout, B. Girau, and F. Rossi. Generic back-propagation in arbitrary feedforward neural networks. In R.F. Albrecht D.W. Pearson, N.C. Steel, editor, *Artificial Neural Nets and Genetic Algorithms - Proc. of ICANNGA*, pages 168–171. Springer-Verlag, 1995.
6. J. Ghosh and K. Hwang. Mapping neural networks onto message-passing multicomputers. *Journal of parallel and distributed computing*, 6:291–330, May 1989.
7. B. Girau. Algorithmes parallèles et modélisations pour les réseaux d'opérateurs. Master's thesis, LIP-ENSL, 1994.
8. B. Girau. Neural network parallelization on a ring of processors : training set partition and load sharing. Research report 94-35, LIP, 1994.
9. B. Girau and H. Paugam-Moisy. Load sharing in the training set partition algorithm for parallel neural learning. In *Proc. IPPS 9^{th} Int. Parallel Processing Symposium*, pages 586–591. IEEE Computer Society Press, 1995.
10. R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Proc. Int. Joint Conf. Neural Networks*, volume 1, pages 593–605, 1989.
11. V. Kumar, S. Shekhar, and M.B. Amin. A scalable parallel formulation of the back-propagation algorithm for hypercubes and related architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1073–1090, October 1994.
12. Y. Le Cun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowsky, editors, *Proc. of the 1988 Connectionist Models Summer School*, pages 21–28. Morgan-Kaufmann, 1988.
13. H. Paugam-Moisy. On a parallel algorithm for back-propagation by partitioning the training set. In *Proc. Neuro-Nîmes*, pages 53–65, 1992.
14. A. Pétrowski. Choosing among several parallel implementations of the backpropagation algorithm. In *Proc. ICNN*, pages 1981–1986, 1994.
15. A. Pétrowski, G. Dreyfus, and C. Girault. Performance analysis of a pipelined back-propagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981, Nov. 1994.
16. H. Yoon and J.H. Nang. Multilayer neural networks on distributed-memory multiprocessors. In *Proc. INNC Paris*, volume 2, pages 669–672, 1990.
17. Q. Zhang and A. Benveniste. Wavelet networks. *IEEE Trans. On Neural Networks*, 3(6):889–898, Nov. 1992.
18. X. Zhang, M. McKenna, J.J. Mesirov, and D.L. Waltz. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, 14:317–327, 1990.