ELSEVIER

Parallel Computing 28 (2002) 861–891

PARALLEL COMPUTING

www.elsevier.com/locate/parco

A scalable parallel algorithm for training a hierarchical mixture of neural experts

Pablo A. Estévez [a], Hélène Paugam-Moisy [b,*], Didier Puzenat [b,c], Manuel Ugarte [a]

[a] *Departamento de Ingeniería Eléctrica, Universidad de Chile, Casilla 412-3, Santiago, Chile*
[b] *Institut des Sciences Cognitives, UMR CNRS 5015, 67 boulevard Pinel, F-69675 Bron Cedex, France*
[c] *Equipe GRIMAAG, Université Antilles-Guyane, Campus de Fouillole, F-97159 Pointe-à-Pitre, France*

Received 7 July 2000; received in revised form 15 September 2001; accepted 10 December 2001

Abstract

Efficient parallel learning algorithms are proposed for training a powerful modular neural network, the hierarchical mixture of experts (HME). Parallelizations are based on the concept of modular parallelism, i.e. parallel execution of network modules. From modeling the speed-up as a function of the number of processors and the number of training examples, several improvements are derived, such as pipelining the training examples by packets. Compared to experimental measurements, theoretical models are accurate. For regular topologies, an analysis of the models shows that the parallel algorithms are highly scalable when the size of the experts grows from linear units to multi-layer perceptrons (MLPs). These results are confirmed experimentally, achieving near-linear speedups for HME-MLP. Although this work can be viewed as a case study in the parallelization of HME neural networks, both algorithms and theoretical models can be expanded to different learning rules or less regular tree architectures.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Artificial neural networks; Parallel learning algorithms; Hierarchical mixture of experts; Expectation-maximization learning algorithm; Scalability

[*] Corresponding author.
E-mail addresses: pestevez@cec.uchile.cl (P.A. Estévez), hpaugam@isc.cnrs.fr (H. Paugam-Moisy), dpuzenat@isc.cnrs.fr (D. Puzenat), mugarte@dcc.uchile.cl (M. Ugarte).

1. Introduction

Although neural networks have inherent massively parallel features, their distributed aspects have been proved difficult to be captured on current parallel computers [7,20,25]. There are several ways to parallelize neural networks, according to the architecture of the network [36], or taking advantage of the matricial learning rule calculations [27], or parallelizing the presentation of examples [22]. Hopp and Prechelt [11] recognize three nested levels of parallelism in neural algorithms: connection parallelism (parallel execution on sets of weights), node parallelism (parallel execution of operations on sets of neurons), and example parallelism (parallel execution of examples on replicated networks). Anyway, it is still a challenge to propose a generic approach providing good performance without special considerations about the specific model of neural networks or the technical characteristics of the parallel computer. For modular neural networks a higher level of parallelism is conceivable, modular parallelism, i.e. parallel execution of network modules. While fine-grain parallel implementations are adequate for deeper levels of parallelism such as connection parallelism, a coarse-grain parallel implementation seems more appropriate for modular parallelism. The main purpose of this article is to emphasize the interest of modular parallelism on the basis of a widely detailed case study in the parallel training of hierarchical mixture of experts (HME) neural networks.

Murre [21] has analyzed the performance of the CALM (categorizing and learning module) learning algorithm for modular neural networks implemented on transputers. Assuming a balanced distribution of modules over processors, Murre showed that the topology of the processor network can strongly reduce data transfer time. Modularity imposes regular constraints on the connectivity that can be used to implement more efficient routing schemes. Even on high massively parallel machines, where a network can be mapped by a one-to-one allocation of neurons to processors, the communications must be optimized, e.g. in [19], Mattes et al. implemented a balanced spanning tree of the interconnection network. Fine-grain parallel implementations can be adequate for deep levels of parallelism such as connection or node parallelism, but a coarse-grain parallel implementation seems to be more appropriate for modular parallelism. Moreover, fine-grain implementations are convenient for specific and regular applications only, such as image processing by simple learning algorithms like Hebbian rule [19].

Most work on parallel neural networks has dealt with the parallelization of the error back-propagation learning algorithm for training multi-layer perceptrons (MLPs) [17,23,26,28,34]. Sudhakar and Murthy [35] have presented a taxonomy of the existing schemes to parallelize the back-propagation algorithm. They classified the parallelization schemes into four categories: network partitioning, pattern partitioning, hybrid partitioning and heuristic partitioning. The network partitioning schemes include both node and connection parallelism [17]. Pattern partitioning divides the pattern set among several processors, either by replicating the network on every processor and each processor works with a subset of the pattern set [2,23] or by pipelining the computation at each layer of the network [27]. Hybrid schemes mix pattern partitioning with network partitioning. Heuristic schemes deal with the neural network graph partitioning for mapping onto specific parallel computer systems. Hendrickson and Leland [9] give a good overview of heuristic methods to generate approximate solutions to graph partitioning. Hendrickson and Kolda [10] survey some recent work on alternative models to the standard approach to graph partitioning.

A survey about multiprocessor simulation of neural networks [24] concludes suggesting the importance of promoting the concept of coarse-grained modular neural networks for an efficient parallel implementation. The present article starts from this point of view and develop a work based on the parallelization of a modular neural network architecture.

From the connectionist point of view also, modular architectures of neural networks offer advantages over a single network in multi-class or multi-task problems [12], especially regarding learning speed, generalization capabilities, and representation capabilities. The HME model [13,15] can discover a recursive decomposition of the input space into nested regions and can learn separate associative mappings within each region. Learning is treated as a maximum likelihood problem. The learning rule most usually applied to the HME model is the expectation-maximization (EM) algorithm [4]. The mixture of experts model has been applied successfully to classification and regression problems [5,14,37–39], including time series prediction [18,40].

A hierarchical mixture of linear experts (with single softmax units as gating networks) run orders of magnitude faster than standard back-propagation networks [15]. More sophisticated expert and gating networks such as MLPs can be required for more complex applications [37,40]. In such conditions, the HME learning phase becomes highly time consuming. This argument is a motivation for studying parallel implementations of HME models. Moreover, since the HME model can be seen as a tree with neural networks at both terminal and nonterminal nodes, its architecture is suitable for a modular, coarse-grain, parallel implementation allocating one subnetwork per processor. Thus our approach is to take advantage of the HME network tree architecture, instead of the computational aspects of the EM algorithm. Distributed versions of the EM algorithm have been developed for specific applications not related to HME models such as positron emission tomography image reconstruction [8].

This article proposes and discusses several versions of parallel learning algorithms for HME models. Starting from measurements for a few processors, with single units as expert and gating networks, we build theoretical models and prove that the proposed parallel algorithms are highly scalable to MLP networks. The running time performance is compared for the sequential and parallel algorithms applied to a regression problem generated with a mixture of Gaussians.

2. HME connectionist model

2.1. The HME architecture

The HME architecture (Fig. 1) is a tree in which the *gating* networks lie at the nonterminal nodes and the *expert* networks lie at the leaves of the tree. The task

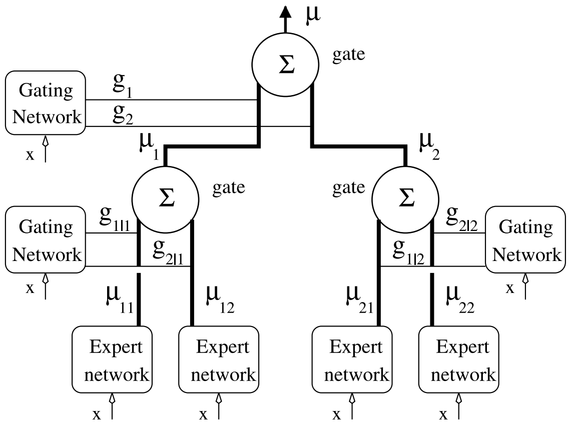

Fig. 1. HME neural network architecture.

of each expert is to approximate a function over a region of the input space. The task of the gating network is to assign the most convenient expert to each input vector. Fig. 1 facilitates to introduce the terminology for a HME tree, on a mixture of four experts, where $\vec{x}$ is the input vector, μ_{ij} is the output (expected value) of the *ij*th expert, $g_i(\vec{x})$ is the output of the top gating network denoting the prior probability for the pattern to be generated by the left or right branch of the root, and $g_{j/i}(\vec{x})$ is the output of the *i*th bottom gating network denoting the prior probability that the pattern is generated by the *ij*th expert. In addition, t is the target (desired output), $P_{ij}(t/\vec{x})$ is the probability associated with the *ij*th expert.

Assuming that experts are mutually exclusive, the overall probability, $P(t/\vec{x})$, and the expected value at the network output, $\mu(\vec{x})$, are given by:

$$P(t|\vec{x}) = \sum_i g_i(\vec{x}) \sum_j g_{j|i}(\vec{x}) P_{ij}(t|\vec{x}),$$

$$\mu(\vec{x}) = \sum_i g_i(\vec{x}) \sum_j g_{j|i}(\vec{x}) \mu_{ij}(\vec{x}),$$

using notations defined for the two-level depth tree shown in Fig. 1, notations that can be easily extended to larger HME networks with a binary tree architecture.

2.2. The EM learning algorithm

Jordan and Jacobs [15] have proposed an EM algorithm for training HME models. EM is an iterative approach to maximum likelihood based on the repetition of two steps: an estimation (E) step and a maximization (M) step. In Jordan and Jacobs's algorithm, the E step consists in calculating the conditional and joint posterior probabilities, which are interpreted as the expected values of missing indicators, using the current values of the parameters. The conditional posterior probabilities at the bottom gating networks $h_{j/i}$, and at the top gating network h_i, are defined respectively as

$$h_{j|i} = \frac{g_{j|i} P_{ij}(t|\vec{x})}{P_i(t|\vec{x})} \quad \text{and} \quad h_i = \frac{g_i \sum_j g_{j|i} P_{ij}(t|\vec{x})}{P(t|\vec{x})}.$$

The joint posterior probabilities are defined as $h_{ij} = h_i h_{j/i}$.

The M step finds the optimal parameters of the model that maximize the expected value of the log-likelihood on the whole data set. The M step reduces to solve a separate maximum likelihood problem for each expert and gating network, applying the iteratively reweighted least-squares algorithm. The targets of the expert networks are the desired outputs of the examples, while the targets of the gating networks are the posterior probabilities.

2.3. The sequential learning algorithm

This section presents our version of the sequential EM algorithm, based on the description given in [15], with the addition of the update rule for the variances proposed in [40]. The variance of the *ij*th expert can be computed as

$$\sigma_{ij}^2 = \frac{\sum_{s=1}^N h_{ij}^s (t^s - \mu_{ij}^s)^2 + \lambda \sigma_{oij}^2}{\sum_{s=1}^N h_{ij}^s + \lambda},$$

where σ_{oij}^2 is a prior variance, and $\lambda \geqslant 0$ is the belief in that prior value. When $\lambda = 0$, the variance of the *ij*th expert is calculated as the weighted average of the squared errors. The weight is given by h_{ij}^s, the joint posterior probability that expert *ij*th generated pattern s, for $s = 1, \ldots, N$. The denominator normalizes the weightings for that expert. The case $\lambda = 0$ corresponds exactly to the maximum likelihood update. When λ takes a large value, the variance of the *ij*th expert is σ_{oij}^2, independent of the data. The need to introduce a prior variance comes from the fact that the maximum likelihood variance update is statistically unreliable when expert *ij* wins only a few patterns.

For each step of the sequential EM algorithm, operations are performed on the whole example set. This version is optimal in the sense that it minimizes the number of forward and backward crossings through the tree. The algorithm is divided into subroutines and described with names of variables corresponding to the two-level tree architecture shown in Fig. 1, with four experts.

1. Randomly initialize the weights for all expert and gating networks.
2. Follow recursively the tree structure, visiting nodes *from bottom to top*:
   - on a terminal node (i.e. a node holding an expert network):
     (a) calculate the outputs $\mu_{ij}(\vec{x})$;
     (b) calculate the Gaussian probabilities $P_{ij}(t|\vec{x})$;
   - on a nonterminal node (i.e. a node holding a gating network):
     (a′) calculate the prior probabilities $g_{j|i}(\vec{x})$ at the bottom gating networks and $g_i(\vec{x})$ at the top gating network (i.e. root node),
     (c) calculate the likelihoods $l_i(t|\vec{x}) = \sum_j g_{j|i}(\vec{x}) P_{ij}(t|\vec{x})$ at intermediate nodes and $P(t|\vec{x})$ at the root node,

(d) calculate the expected values (i.e. outputs) $\mu_i(\vec{x})$ at intermediate nodes and $\mu(\vec{x})$ at the root node.

3. Follow recursively the tree structure, visiting nodes *from top to bottom*:
   - on a nonterminal node (gating network):
     (e) calculate the conditional posterior probabilities: $h_{j|i}$ and h_i,
     (e′) calculate the joint posterior probabilities: $h_{ij} = h_i h_{j|i}$,
     (g) update the weights V of the current gating network;
   - on a terminal node (expert network):
     (f) update the variances σ^2_{ij},
     (g) update the weights W of the current expert network;
   - on the root node only:
     (h) compute a stop criterion and go back to step 2, unless the criterion is satisfied.

2.4. HME with linear experts

First consider a hierarchical mixture of linear experts in which the experts are single linear units computing a weighted sum of their inputs, whereas the gates are single softmax units. The weight update step implies to solve a set of independent linear systems by Cholesky decomposition, one for each expert and gating network [15]. If ∇L denotes the gradient of the log-likelihood, $\Delta\Theta$ is the weight adjustment vector (Θ is the vector of free parameters of the corresponding network) and F is minus the expected value of the Hessian matrix, then the linear systems respect the equation $F\Delta\Theta = \nabla L$. It can be shown that $F = X^{\mathrm{t}}PX$, where X^{t} is the transposed input matrix and P is a diagonal matrix. Moreover the gradient of the log-likelihood is $X^{\mathrm{t}}P\delta$, where P is the above mentioned diagonal matrix and δ is an expression that depends on the local error.

In a more general HME model, every node is a MLP which develops a classical back-propagation learning algorithm [32]. The case of MLPs as expert and gating networks will be considered later in Section 7.

3. Parallel algorithm

Efficient parallel implementations of neural networks are often based on a decomposition of the example set into blocks, which are learned simultaneously on different processors, and thus modifying the behavior of the training algorithm. Usually they take advantage of the robustness of the neural learning processes, e.g. back-propagation learning [23,26,28,33], Kohonen self-organizing maps [3], prototype-based incremental classifier [31], in order to perform as well as the sequential algorithm, and faster, although in a different way. On the contrary, we propose a parallel algorithm, based on a modular architectural decomposition, which respects exactly the behavior of the sequential EM algorithm. The parallel algorithm has a coarse granularity and assumes that the computation to be realized between two consecutive communications is sufficiently expensive.

The EM algorithm presented in the previous section supports different types of stopping criteria (e.g. number of iterations, error level, etc.). The choice should be specified for each application and set of experiments. For all our experiments, a number of iterations has been fixed. As the error only depends on the network architecture and the number of learning iterations, this strategy allows us to check the correctness of each parallel implementation: for an equally sized tree, the sequential and parallel final errors must exactly match.

3.1. Mapping the HME on a network of processors

The computational tasks are decomposed into two types of modules: each expert module is in charge of an expert neural network, and each gating module is in charge of both a gating neural network and the computation done at the corresponding node of the tree. This strategy yields an efficient mapping, with highly interconnected parts of the network mapped onto a same processor, in order to reduce the number of communications.

For a binary tree with n_e experts at the leaves, $2 \times n_e - 1$ processors are required to implement one module per processor. Fig. 2 shows the mapping onto seven processors and the communication paths for a complete binary tree with four experts.

3.2. Parallel version of the learning algorithm

The parallel program starts by initializing the mapping and the modules. The $2 \times n_e - 1$ processes (expert and gating modules) are spawned recursively on the processors of the parallel machine. Each module reads the whole training set and initializes its weights randomly. The main loop of the algorithm (a cycle) consists of three parts: *expert module algorithm*; *inner-level gating module algorithm*; and *root gating*

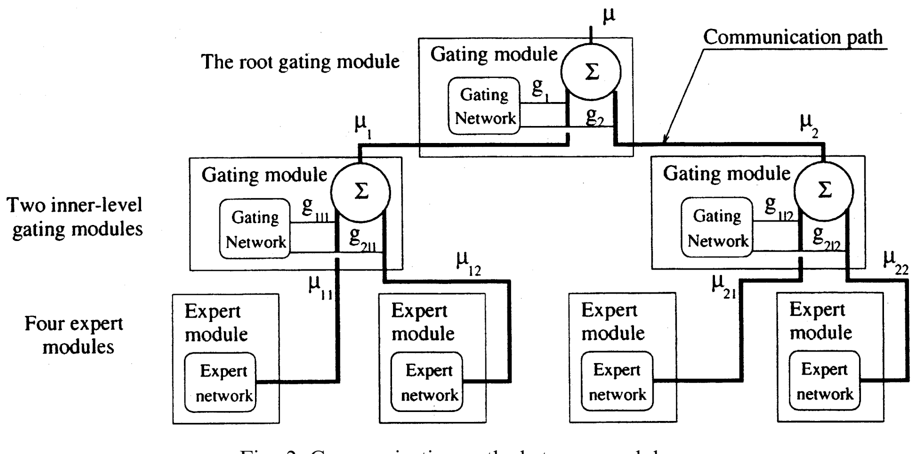

Fig. 2. Communication paths between modules.

module algorithm. When the stop criterion is satisfied, a message is sent by the root module to all the other modules, recursively in the tree, and all the modules halt. We describe below the mapping of the subroutines and the intermediate data communications of the parallel learning algorithm, for each type of module, with the notations introduced for describing the sequential learning algorithm.

For all the expert modules: Apply each of the following steps to all the examples in the training set:

(a) calculate the outputs $\mu_{ij}(\vec{x})$;
(b) calculate the Gaussian probabilities $P_{ij}(t/\vec{x})$;
$\rightleftharpoons$ *send* the Gaussian probabilities and outputs to its parent in the tree;
$\rightleftharpoons$ *receive* the joint posterior probabilities from its parent in the tree;
(f) update the variances σ_{ij}^2;
(g) update the weights W.

For all the inner-level gating modules: Apply each of the following steps to all the examples in the training set:

(a′) calculate the prior probabilities $g_{j/i}(\vec{x})$;
$\rightleftharpoons$ receive the likelihoods from its two children (in a binary tree);
(c) calculate the new likelihood $l_i(t|\vec{x}) = \sum_j g_{j|i}(\vec{x})P_{ij}(t|\vec{x})$;
$\rightleftharpoons$ send the likelihood to its parent;
(e) calculate the conditional posterior probabilities $h_{j/i}$;
$\rightleftharpoons$ receive the outputs from its two children;
(d) calculate the outputs (i.e. expected values) $\mu_i(\vec{x})$;
$\rightleftharpoons$ send the outputs to its parent;
$\rightleftharpoons$ receive the conditional posterior probabilities from its parent;
(e′) calculate the joint posterior probabilities $h_{ij} = h_i h_{j/i}$;
$\rightleftharpoons$ send the joint posterior probabilities to its two children;
(g) update the weights V_1;

For the gating module at the root of the tree: Apply each of the following steps to all the examples in the training set:

(a′) calculate the prior probabilities $g_i(\vec{x})$
$\rightleftharpoons$ receive the likelihoods from its two children;
(c) calculate the new likelihood $P(t/\vec{x})$;
(e) calculate the conditional posterior probabilities h_i;
$\rightleftharpoons$ send the conditional posterior probabilities to its two children;
(g) update the weights V_2;
$\rightleftharpoons$ receive the outputs from its two children;
(d) calculate the global output $\mu(\vec{x})$;
(h) calculate the error and evaluate the stop criterion;
continue until the criterion is satisfied (or send *stop* messages).

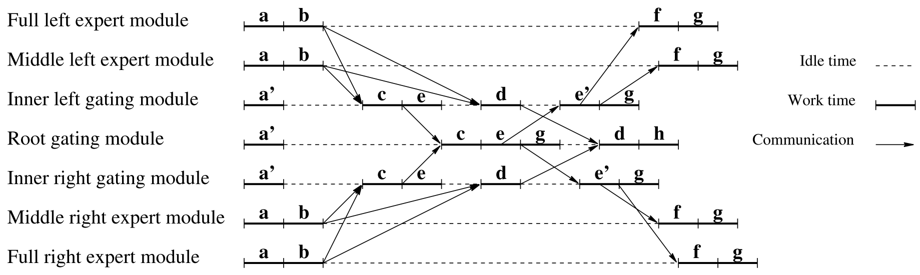

Fig. 3. Schematic temporal diagram (linear units).

3.3. Scheduling the computation and intermediate data communications

A deep analysis of the sequential algorithm described in Section 2.3, has highlighted the feasibility of delaying receptions of intermediate outputs until data are absolutely needed, in order to overlap communications and computations [6]. The scheduling of this algorithm can be represented on a temporal diagram, with one line per processor, and for one cycle long.

Fig. 3 exemplifies a temporal diagram for the case of a binary tree with four experts. Such a temporal diagram is a schematic diagram only, since the computation times of all steps of the algorithm are represented as being equal, which is not realistic. However, this theoretical time diagram is useful as a basis for achieving further improvements. Experimental time diagrams will be presented in following sections.

3.4. The test problem

In order to measure the running times of both the sequential program and the parallel one, the approximation of a mixture density of two Gaussians has been considered. The mixture is denoted by:

$$\alpha_1\mathcal{N}(\mu_1,\sigma_1^2)+\alpha_2\mathcal{N}(\mu_2,\sigma_2^2), \tag{1}$$

where μ_i is the mean, σ_i^2 is the variance of the ith Gaussian $\mathcal{N}(\mu_i,\sigma_i^2)$ and the α_i are the mixture coefficients. In this problem experts have to learn the parameters μ_i and σ_i^2, while the gating networks have to learn the mixture proportions α_i. The purpose of testing the program on this mixture of Gaussians is to obtain experimental time measurements and to collect information on the behavior of the speedup performance both as a function of the number of processors and the size of the example set. Both sequential and parallel algorithms have the same learning behavior and consequently the same performance, for a fixed tree architecture, hence the performance of the HME algorithm, in terms of the quality of approximation of the mixture of Gaussians, will not be addressed.

4. Parallel implementation of HME with linear experts

The parallel program has been implemented at the University of Chile, on a Matra Capitan computer. The Matra Capitan is a MIMD parallel computer. In order to make the Capitan scalable, the topology is a ring of "hyper-nodes" capable of 50 Mb/s bidirectional communications. Each hyper-node is a cluster of six nodes. The six processors within a hyper-node are on the same bus, capable of 40 Mb/s transfer. Each processor has its own private memory, but a small part is shared and used as buffers by "CapNet": the communication software from Matra. However, the portable "message passing interface" (MPI) communication library has been used instead of CapNet, in order to make the software (written in C) portable to other MIMD computers with distributed memory, as well as to clusters of workstations. Indeed, MPI, using CapNet, hides low-level mechanisms from the C program [30]. However, using MPI over CapNet, it is not possible to allocate a specific processor to a given task, which makes precise time measurements difficult since intra-hyper-node and inter-hyper-node communication times are not equal. Hopefully, thanks to the deterministic placement algorithm of MPI, a given HME architecture usually leads to a specific placement and to accurate results.

Experiments have been performed on a machine which has two hyper-nodes, with 12 "Micro-Sparc II" processors running at 85 MHz, with 32 MB of memory. One more node, a "service node", running the SUN Solaris Operating System, undertakes to compile programs and to initialize and control the machine.

4.1. Experimental time measurements

First experiment: Two different architectures of the HME network were simulated: (i) two experts and one gating modules with three processors, and (ii) four experts and three gating modules with seven processors. A large set of 10,000 examples was randomly generated and the networks were trained on subsets of data of different sizes. Fig. 4 shows how the speedup grows with the sample size, saturating quickly at a maximum value of 2.6 with three processors and 4.5 with seven processors.

Second experiment: The size of the dataset was fixed to 8000 examples and the number of processors varied from 3 to 11. Note that the HME architecture is no longer a complete tree since the number of processors is not equal to $2^n - 1$ for an integer value of n. Fig. 5 presents the speedup as a function of the number of processors, for a fixed sample size.

For the parallel learning algorithm described in Section 3.2, the speedup grows linearly (up to 11 processors), but it is far from being equal to the number of processors, due to communication overhead and idle time (i.e. when a processor is wasting time, waiting for a message). However, a speedup of 6.6 for 11 processors is cheering, compared to several other parallel learning algorithms for neural networks. The main explanation for idle time is that the current algorithm is based on the tree architecture of the HME model. The tree must be traversed once from bottom to top and once from top to bottom, at each cycle. Hence, expert modules and even inter-

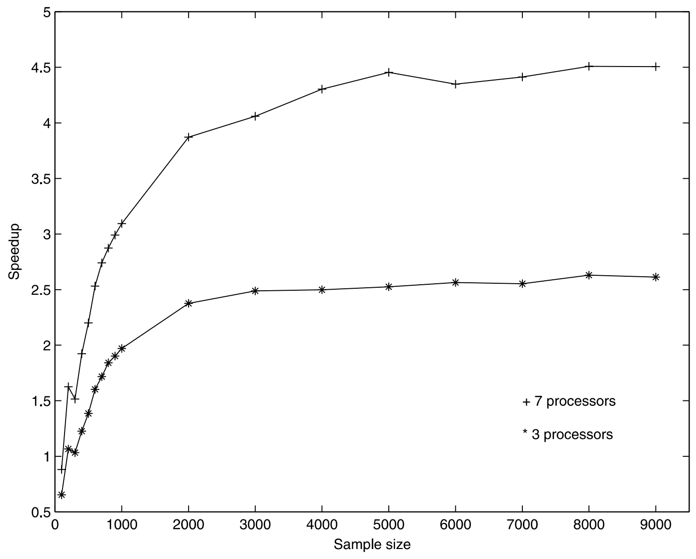

Fig. 4. Speedup as a function of the sample size (linear units).

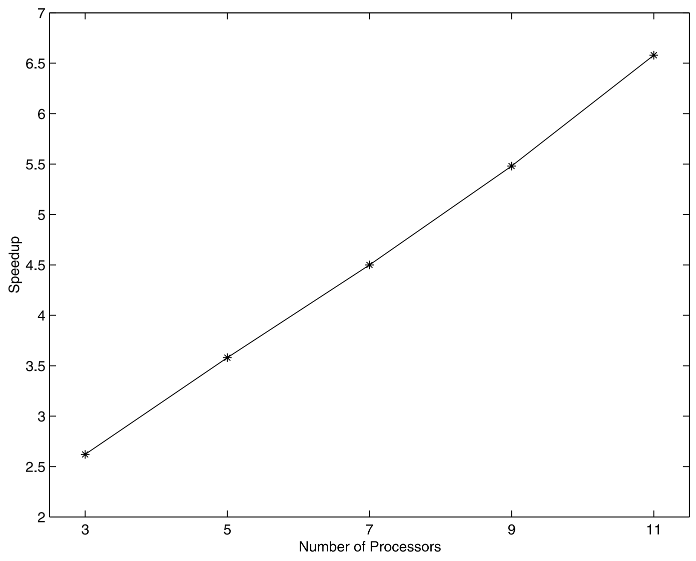

Fig. 5. Speedup as a function of the number of processors (linear units).

mediate gating modules must wait for results computed by their children and parent modules in the tree, alternatively.

Fig. 6 shows an experimental time diagram for a seven node tree, for one cycle of the parallel learning algorithm and has to be compared to the schematic temporal

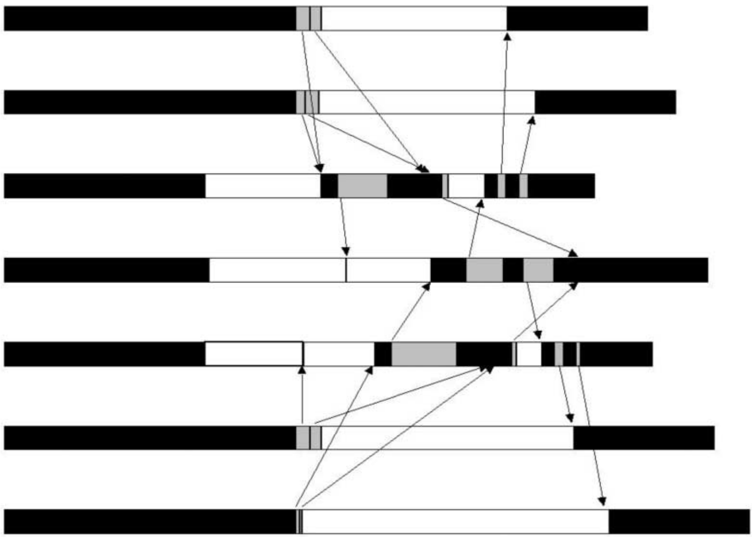

Fig. 6. Experimental time diagram (linear units). Load of each processor is shown in black, message sending is depicted in grey and idle time in white.

diagram of Fig. 3. Load of each processor is shown in black, message sending is depicted in grey and idle time in white. Fine arrows link the starting time of an emission to the end reception time on the receiving processor.

In order to explain the experimental results and to predict the behavior of the parallel algorithm for higher numbers of processors or larger sample sizes, it is necessary to build a detailed model of the temporal behavior of the parallel algorithm. Moreover, this modeling will be applied for proving, in a further section, that the same algorithm scales up nicely when MLPs are used as expert and gating networks, instead of single units.

Note that the actual communication time is different for inter-hyper-node and intra-hyper-node communications. It also depends on the load of the communication network of the parallel machine, i.e. on the number of simultaneous communications. Hence the experimental communication time can vary slightly, even for constant length messages, which is hard to be modeled. These variations will not be taken into account in the theoretical models presented in further sections.

5. Modeling

A formal expression for the running time of the longest path can be derived from the temporal diagram of Fig. 3. The running time can be expressed as a sum of calculation times T_k(subroutine), and communication times T_{m} which are assumed equal for all the messages (as in Fig. 3). The following notation is introduced: $A = \mathrm{a} + \mathrm{b} + \mathrm{f} + \mathrm{g}$ is the concatenation of all the subroutines computed by each expert module and $B = \mathrm{a}' + \mathrm{c} + \mathrm{d} + \mathrm{e} + \mathrm{g}$ corresponds to the common computation at all the gating modules. The latter excludes the noncommon tasks, e.g. the step (h), which is computed only at the root-level gating module (calculation of the overall error of the HME network and evaluation of the stop criterion).

5.1. Speedup as a function of the number of processors

For a general binary tree of depth $H \geqslant 1$, with n_e expert modules at the leaves, implemented on n_p processors, the run time T_seq of the sequential algorithm described in Section 2.3, and the run time T_par of the parallel algorithm described in Section 3.2, are as follows:

$$T_\mathrm{seq} = n_\mathrm{e} T_k(A) + (n_\mathrm{e} - 1)T_k(B) + (n_\mathrm{e} - 2)T_k(\mathrm{e}') + T_k(\mathrm{h}), \tag{2}$$

$$T_\mathrm{par} = T_k(A) + HT_k(\mathrm{c}) + (H-1)T_k(\mathrm{e}') + T_k(\mathrm{e}) + 2n_\mathrm{e}T_\mathrm{m}. \tag{3}$$

Note that the relationship between the number of processors and the number of expert modules in a binary tree is $n_\mathrm{p} = 2 \times n_\mathrm{e} - 1$. Making use of this relationship and rearranging the terms in Eqs. (2) and (3), the following expressions for T_seq and T_par, as a function of the number of processors, are obtained:

$$T_\mathrm{seq} = a_0 + a_1 \times n_\mathrm{p}, \tag{4}$$

where

$$\begin{cases} a_0 = \frac{1}{2}\left(T_k(A) - T_k(B) - 3 \times T_k(\mathrm{e}')\right) + T_k(\mathrm{h}), \\ a_1 = \frac{1}{2}\left(T_k(A) + T_k(B) + T_k(\mathrm{e}')\right), \end{cases}$$

$$T_\mathrm{par} = b_0 + b_{01} \times H + b_1 \times n_\mathrm{p}, \tag{5}$$

where

$$\begin{cases} b_0 = T_k(A) - T_k(\mathrm{e}') + T_k(\mathrm{e}) + T_\mathrm{m}, \\ b_{01} = T_k(\mathrm{c}) + T_k(\mathrm{e}'), \\ b_1 = T_\mathrm{m}. \end{cases}$$

In a binary tree, the number of processors n_p is lower bounded by $2^H + 1$ and upper bounded by $2^{H+1} - 1$, i.e. $H = \mathrm{O}(\log_2 n_\mathrm{p})$.

The speedup S can be expressed as a function of the *overhead function* $T_\mathrm{o} = n_\mathrm{p}T_\mathrm{par} - T_\mathrm{seq}$, as defined in [16]:

$$S = \frac{T_\mathrm{seq}}{T_\mathrm{par}} = \frac{n_\mathrm{p}}{1 + \frac{T_\mathrm{o}}{T_\mathrm{seq}}}. \tag{6}$$

From its definition, the overhead function includes the communication overhead and the idle time of all processors. From our modeling, the overhead function is as follows:

$$T_\mathrm{o} = k_0 + k_1 \times n_\mathrm{p} + k_{12} \times H \times n_\mathrm{p} + k_2 \times n_\mathrm{p}^2, \tag{7}$$

where

$$\begin{cases} k_0 = -\frac{1}{2}\left(T_k(A) - T_k(B) - 3 \times T_k(\mathrm{e}')\right) - T_k(\mathrm{h}), \\ k_1 = \frac{1}{2}\left(T_k(A) - T_k(B) - 3 \times T_k(\mathrm{e}')\right) + T_k(\mathrm{e}) + T_{\mathrm{m}}, \\ k_{12} = T_k(\mathrm{c}) + T_k(\mathrm{e}'), \\ k_2 = T_{\mathrm{m}}. \end{cases}$$

The speedup can be calculated from (4), (6) and (7). The theoretical model indicates that the speedup cannot exceed a maximum value given by an asymptotic approximation of its expression, for n_{p} increasing towards infinity:

$$\lim_{n_{\mathrm{p}}\to\infty} S(n_{\mathrm{p}}) \approx \frac{a_1}{b_1} = \frac{T_k(A) + T_k(B) + T_k(\mathrm{e}')}{2 \times T_{\mathrm{m}}},$$

quantity strongly dependent on the ratio between computation and communication times which is available on the parallel computer.

5.2. Speedup as a function of the sample size

For each communication phase, we assume that the size of the message is linear in the number of examples N, i.e. the sample size. Hence $T_{\mathrm{m}} = T_{\mathrm{s}} + N \times T'_{\mathrm{m}}$, where T_{s} is the startup time of the communication and T'_{m} the transfer time for elementary real data. Likewise, for each calculation phase the computation times are modeled as linear in the sample size. For each subroutine $T_k(\mathrm{sub}) = \lambda_{\mathrm{init}}(\mathrm{sub}) + N \times T'_k(\mathrm{sub})$, where λ_{init} is an initialization time and T'_k is the calculation time of elementary real data. Thus the computation times in (4), (5) and (7) can be expressed as $a_i = \lambda_{a_i} + N \times a'_i$, $b_i = \lambda_{b_i} + N \times b'_i$, and $k_i = \lambda_{k_i} + N \times k'_i$, respectively, where the λ_{index} are constants. For very large sample sizes, asymptotic behavior is obtained when N goes to infinity:

$$\lim_{N\to+\infty} S(N) \approx \frac{a'_0 + a'_1 n_{\mathrm{p}}}{b'_0 + b'_{01}H + b'_1 n_{\mathrm{p}}}.$$

The limit enhances the fact that the speedup saturates at a maximum value, which depends on the number of processors, when the sample size grows towards infinity. This result confirms the experimental speedup curves shown in Fig. 4.

5.3. Theoretical speedup vs. experimental speedup

Measurements of experimental speedup were performed on a HME network of four experts, mapped onto seven processors, for learning a mixture of two Gaussians from a data set of 8000 examples. The calculation times, in seconds, were measured separately for all the phases specified by letters in the temporal diagram of Fig. 3. The measurements were averaged over several cycles. In all cases the standard deviation was less than 0.001. The calculation times are as follows: $T_k(A) = 0.247$, $T_k(B) = 0.194$, $T_k(\mathrm{c}) = 0.009$, $T_k(\mathrm{e}) = 0.022$, $T_k(\mathrm{e}') = 0.014$, and $T_k(\mathrm{h}) = 0.040$. The communication time T_{m}, in seconds, is harder to be modeled since the distribution of measurements is not Gaussian (the mean value 0.009 is much higher than the median 0.006). The following range of values for the communication time has been considered: $T_m = 0.009 \pm 0.005$, which covers about 60% of the cases. The constants defined in the modeling were estimated as follows (for $T_m = 0.009$ s):

- for Eq. (4): $a_0 = 0.044$, $a_1 = 0.228$,
- for Eq. (5): $b_0 = 0.262$, $b_{01} = 0.024$, $b_1 = 0.009$,
- for Eq. (7): $k_0 = -0.044$, $k_1 = 0.035$, $k_{12} = 0.024$, $k_2 = 0.009$.

From these values, the limit of the speedup, when the number of processors goes to infinity, is $a_1/b_1 = 25$.

Fig. 7 shows both the theoretical speedup (the line with error bars including the deviation of the communication time from the mean) and the experimental speedup (x-mark) as a function of the number of processors. The theoretical estimation is quite accurate. All the experimental values are within the error bars, except for the case of three processors where the estimation is slightly pessimistic. This result suggests that the average communication time for three processors is faster than with seven processors, since the measurements have been based on the latter configuration. Explanations for this behavior can be the difference between intra-hyper-node and inter-hyper-node communication times, as well as the increasing (with n_p) number of simultaneous communications.

Next, we compare theoretical speedup and experimental speedup as a function of the sample size. The experimental communication and calculation times, per elementary data, as well as the initialization times of each subroutine, were estimated from

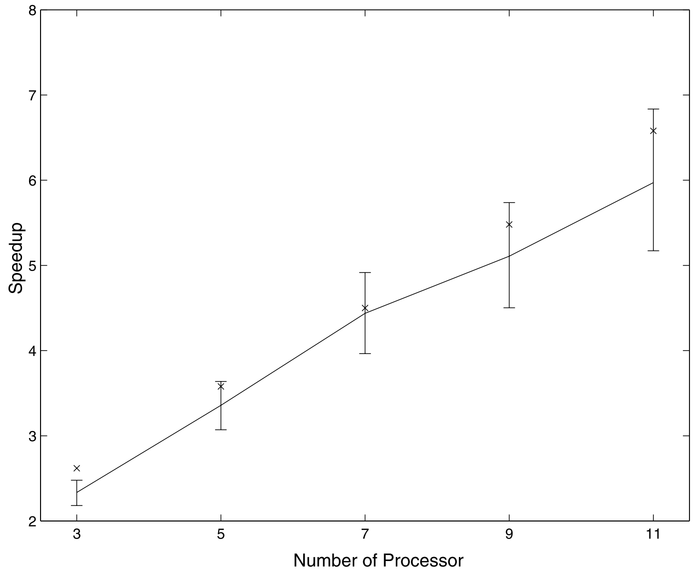

Fig. 7. Theoretical (—) and experimental (× marks) speedups.

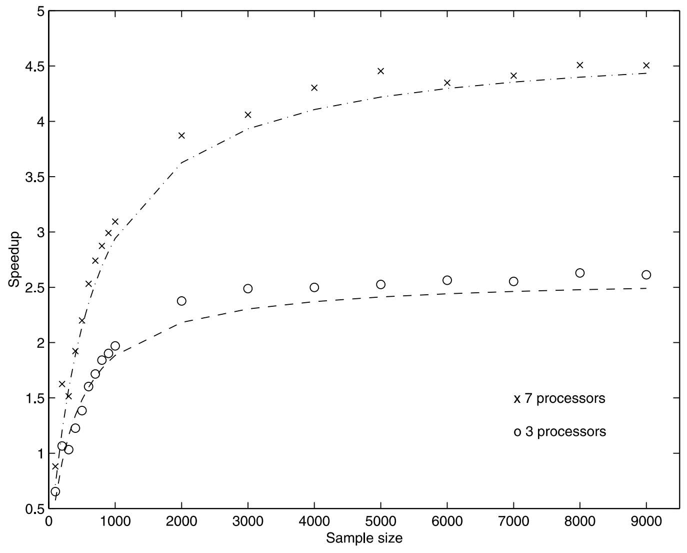

Fig. 8. Theoretical (- - -) and experimental (○ and × marks) speedups as a function of sample size.

running a HME network of four experts and three gating modules on seven processors, learning from 2000, 4000 and 8000 data. The calculation times were measured for all the phases specified by letters in the temporal diagram of Fig. 3. The measurements were averaged over several cycles. The calculation times, in milliseconds, were as follows: $T'_k(A) = 0.031$, $T'_k(B) = 0.025$, $T'_k(\mathrm{c}) = 0.002$, $T'_k(\mathrm{e}) = 0.003$, $T'_k(\mathrm{e}') = 0.002$, and $T'_k(\mathrm{h}) = 0.005$. The initialization times were as follows: $\lambda_A = 0.190$, $\lambda_\mathrm{B} = 0.272$, $\lambda_\mathrm{c} = 0.067$, $\lambda_\mathrm{e} = 0.150$, $\lambda_{\mathrm{e}'} = 0.110$, and $\lambda_\mathrm{h} = 0.077$. The startup time per communication was $T_\mathrm{s} = 3.3$ ms. The range of elementary transfer times corresponding to the range of values taken above for the communication times is, in microseconds, $T'_\mathrm{m} \in [0.09, 1.34]$.

Fig. 8 shows both the theoretical speedup (the lines) and the experimental speedup (x- and -o marks) as a function of sample size. The experimental curves are the same as in Fig. 4. Due to the variation of communication time for three and seven processors, different transfer times were applied to each calculation: $T'_\mathrm{m} = 0.09$ μs for three processors and $T'_\mathrm{m} = 0.7$ μs for seven processors. From these values, the limit of the speedup, when the sample size goes to infinity, is $S = 2.60$ for three processors and $S = 4.74$ for seven processors, which well matches the results in Fig. 4.

6. Pipelining packets of examples

6.1. Principle and model

A way to improve the speedup is to pipeline the examples by packets in order to overlap calculations and communications. A calculation cycle in a gating module can start with the first examples, even if the computations of the expert module are not achieved for the whole training set. Since it would be too much time consuming to send intermediate results after each example, the messages are split into a few number of packets. Both along the bottom-up and the top-down paths, each packet of results is sent as soon as calculated, hence the next module in the tree can start its computation earlier. Thus the overall idle time is reduced. The parallel time for a p-packet pipeline is denoted by T^p_{par}. For the degenerated case $p = 1$, T^1_{par} corresponds to the time of the version without pipelining, given by (5). This expression can be decomposed in two terms: the computation $T_k(A)$ performed by expert modules, and the computation $T_k(R)$ performed on other processors, plus the communication time:

$$T^1_{\mathrm{par}} = T_k(A) + T_k(R) + (n_{\mathrm{p}} + 1)T_{\mathrm{m}} = T_k(A) + 2T_{\mathrm{m}} + \underbrace{T_k(R) + (n_{\mathrm{p}} - 1)T_{\mathrm{m}}}_{\text{Idle time for experts}}, \tag{8}$$

where $T_k(R) = H(T_k(\mathrm{c}) + T_k(\mathrm{e}')) + T_k(\mathrm{e}) - T_k(\mathrm{e}')$.

The purpose of the pipelining is to reduce as much as possible the idle time of a processor holding an expert module, $T_k(R) + (n_{\mathrm{p}} - 1)T_{\mathrm{m}}$, thus shortening the critical path. Fig. 9 illustrates a schematic temporal diagram of the critical path, for a seven nodes tree, and a two-packet pipeline. Steps presented in italic are performed on the second packet of examples. On the expert modules, in the p-packet algorithm, the whole steps (a) and (b) must be over before starting the first part of steps (f) and (g).

The communication time for the p-packet algorithm is modeled as $T^p_{\mathrm{m}} = T_{\mathrm{s}} + (1/p)NT'_{\mathrm{m}}$, where N is the sample size, $p \geqslant 2$ is the number of packets. The parallel running time is modeled as:

$$T^p_{\mathrm{par}} = T_k(A) + 2pT^p_{\mathrm{m}} + (p-1)\max\left(0, T^p_{\mathrm{cp}} - \frac{1}{p}(T_k(\mathrm{a}) + T_k(\mathrm{b}))\right) + \max\left(0, T^p_{\mathrm{cp}} - \frac{p-1}{p}(T_k(\mathrm{f}) + T_k(\mathrm{g}))\right), \tag{9}$$

where T^p_{cp} is the critical path time:

$$T^p_{\mathrm{cp}} = \frac{1}{p}T_k(R) + (n_{\mathrm{p}} - 1)T_{\mathrm{s}} + \frac{1}{p}(n_{\mathrm{p}} - 1)NT'_{\mathrm{m}}.$$

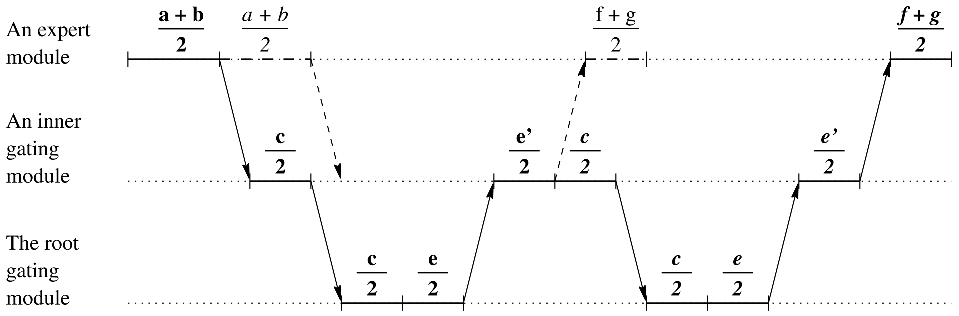

Fig. 9. Critical path diagram for the two-packets version (linear units).

From (9) it is easy to see that four cases can occur, depending on which arguments of the maximum functions take the maximum values. Note that $T_k(\mathrm{a}) + T_k(\mathrm{b})$ is greater than $T_k(\mathrm{f}) + T_k(\mathrm{g})$, for all sample sizes, in the experimental conditions of this article. For this reason the computation of the term $T_k(\mathrm{a}) + T_k(\mathrm{b})$ is split in p portions, while the calculation of term $T_k(\mathrm{f}) + T_k(\mathrm{g})$ is done in two steps. Note that all communications convey packets of size N/p.

Case I: $T^p_{\mathrm{cp}} < ((p-1)/p)(T_k(\mathrm{f}) + T_k(\mathrm{g}))$ and $T^p_{\mathrm{cp}} < (1/p)(T_k(\mathrm{a}) + T_k(\mathrm{b}))$

In this case, $T^p_{\mathrm{par}} = T_k(A) + 2pT^p_{\mathrm{m}}$, i.e. computations end on time and experts do not waste time. From this last expression and (4), the overhead of the p-packet pipeline can be computed as:

$$T^p_{\mathrm{o}} = k_0 + k'_1 n_{\mathrm{p}}, \tag{10}$$

where

$$k'_1 = \frac{1}{2}(T_k(A) - T_k(B) - T_k(\mathrm{e}')) + 2T_{\mathrm{m}} + 2(p-1)T_{\mathrm{s}}, \tag{11}$$

and k_0 is the same coefficient as in (7). The coefficient k_0 and the first term on the right side of k'_1 in (11) represent the idle time in the gating modules, which cannot be reduced with the p-packet approach.

Note that the overhead function (7) of the one-packet algorithm is a quadratic function of the number of processors, while the overhead function (10) for the p-packet algorithm in Case I is linear in n_{p}. The speedup for the p-packet algorithm can be calculated from (4), (6) and (10). The theoretical model indicates that the speedup will be almost linear for n_{p} sufficiently large:

$$S^p \approx \frac{n_{\mathrm{p}}}{\left(1 + \frac{k'_1}{a_1}\right)}.$$

Case II: $T^p_{\mathrm{cp}} > (1/p)(T_k(\mathrm{a}) + T_k(\mathrm{b}))$ and $T^p_{\mathrm{cp}} > ((p-1)/p)(T_k(\mathrm{f}) + T_k(\mathrm{g}))$

In this case:

$$\begin{aligned} T^p_{\mathrm{par}} &= T_k(A) + 2pT^p_{\mathrm{m}} + pT^p_{\mathrm{cp}} - \frac{p-1}{p}(T_k(\mathrm{a}) + T_k(\mathrm{b}) + T_k(\mathrm{f}) + T_k(\mathrm{g})) \\ &= T_k(A) + 2pT^p_{\mathrm{m}} + pT^p_{\mathrm{cp}} - \frac{p-1}{p}T_k(A) \\ &= \frac{1}{p}T_k(A) + 2pT^p_{\mathrm{m}} + T_k(R) + p(n_{\mathrm{p}} - 1)T_{\mathrm{s}} + (n_{\mathrm{p}} - 1)NT'_{\mathrm{m}} \\ &= \frac{1}{p}T_k(A) + T_k(R) + p(n_{\mathrm{p}} + 1)T_{\mathrm{s}} + (n_{\mathrm{p}} + 1)NT'_{\mathrm{m}} \\ &= \frac{1}{p}T_k(A) + T_k(R) + (p-1)(n_{\mathrm{p}} + 1)T_{\mathrm{s}} + (n_{\mathrm{p}} + 1)T_{\mathrm{m}}. \end{aligned} \tag{12}$$

In order to measure the gain of the pipelined version, T^p_{par} has to be compared to T^1_{par}, i.e., (8) and (12). The speedup is increased if the difference $T^1_{\mathrm{par}} - T^p_{\mathrm{par}}$ is positive, i.e. when:

$$\frac{1}{p}T_k(A) - (n_\mathrm{p}+1)T_\mathrm{s} > 0. \tag{13}$$

Eq. (13) implies that there is a critical sample size (a lower bound N_critic) for obtaining a speedup gain. From $T_k(A) = \lambda_A + NT'_k(A)$ and (13), we obtain:

$$N_\mathrm{critic} = \frac{p(n_\mathrm{p}+1)T_\mathrm{s} - \lambda_A}{T'_k(A)} \approx \frac{p(n_\mathrm{p}+1)T_\mathrm{s}}{T'_k(A)}. \tag{14}$$

Case III: $(1/p)(T_k(\mathrm{a}) + T_k(\mathrm{b})) > T^p_\mathrm{cp} > ((p-1)/p)(T_k(\mathrm{f}) + T_k(\mathrm{g}))$
In this case:

$$\begin{aligned} T^p_\mathrm{par} &= T_k(A) + 2pT^p_\mathrm{m} + T^p_\mathrm{cp} - \frac{p-1}{p}(T_k(\mathrm{f}) + T_k(\mathrm{g})) \\ &= T_k(A) + 2T_\mathrm{m} + 2(p-1)T_\mathrm{s} + T^p_\mathrm{cp} - \frac{p-1}{p}(T_k(\mathrm{f}) + T_k(\mathrm{g})). \end{aligned} \tag{15}$$

From (8) and (9), T^1_par can be rearranged as follows:

$$T^1_\mathrm{par} = T_k(A) + pT^p_\mathrm{cp} - (p-1)(n_\mathrm{p}-1)T_\mathrm{s} + 2T_\mathrm{m}. \tag{16}$$

The speedup of the p-packet version is increased with respect to the one-packet version if the difference $T^1_\mathrm{par} - T^p_\mathrm{par}$ is positive:

$$\begin{aligned} &(p-1)T^p_\mathrm{cp} + \frac{p-1}{p}(T_k(\mathrm{f}) + T_k(\mathrm{g})) - (p-1)(n_\mathrm{p}+1)T_\mathrm{s} > 0, \\ &T^p_\mathrm{cp} - \frac{1}{p}(T_k(\mathrm{a}) + T_k(\mathrm{b})) + \frac{1}{p}T_k(A) - (n_\mathrm{p}+1)T_\mathrm{s} > 0. \end{aligned} \tag{17}$$

This implies that $(1/p)T_k(A) - (n_\mathrm{p}+1)T_\mathrm{s} > \epsilon_1 > 0$, where $\epsilon_1 = (1/p)(T_k(\mathrm{a}) + T_k(\mathrm{b})) - T^p_\mathrm{cp} > 0$ due to the condition of Case III. The former inequality implies that (13) must be satisfied, therefore the sample size should be at least N_critic. A similar expression to (14) can be easily deduced for this case.

Case IV: $((p-1)/p)(T_k(\mathrm{f})) + T_k(\mathrm{g}) > T^p_\mathrm{cp} > (1/p)(T_k(\mathrm{a}) + T_k(\mathrm{b}))$
In this case:

$$\begin{aligned} T^p_\mathrm{par} &= T_k(A) + 2pT^p_\mathrm{m} + (p-1)T^p_\mathrm{cp} - \frac{p-1}{p}(T_k(\mathrm{a}) + T_k(\mathrm{b})) \\ &= T_k(A) + 2T_\mathrm{m} + 2(p-1)T_\mathrm{s} + (p-1)T^p_\mathrm{cp} - \frac{p-1}{p}(T_k(\mathrm{a}) + T_k(\mathrm{b})). \end{aligned} \tag{18}$$

Subtracting (18) from (16), we obtain the condition for a speedup gain:

$$\begin{aligned} &T^p_\mathrm{cp} + \frac{p-1}{p}(T_k(\mathrm{a}) + T_k(\mathrm{b})) - (p-1)(n_\mathrm{p}+1)T_\mathrm{s} > 0, \\ &\frac{1}{p-1}T^p_\mathrm{cp} - \frac{1}{p}(T_k(\mathrm{f}) + T_k(\mathrm{g})) > (n_\mathrm{p}+1)T_\mathrm{s} - \frac{1}{p}T_k(A). \end{aligned} \tag{19}$$

This implies that $(1/p)T_k(A) - (n_\mathrm{p}+1)T_\mathrm{s} > \epsilon_2 > 0$ where $\epsilon_2 = (1/p)(T_k(\mathrm{f}) + T_k(\mathrm{g})) - (1/(p-1))T^p_\mathrm{cp} > 0$ due to the condition of Case IV. As in Case III, the former inequality implies that (13) must be satisfied, therefore the sample size should be at least N_{critic}.

The condition of Case I implies approximately that:

$$n_{\text{p}} \leqslant \frac{\min(T'_k(\text{a}) + T'_k(\text{b}), (p-1)(T'_k(\text{f}) + T'_k(\text{g}))) - T'_k(R)}{(p/N)T_{\text{s}} + T'_{\text{m}}} + 1. \tag{20}$$

Likewise, the condition of Case II implies approximately that:

$$n_{\text{p}} \geqslant \frac{\max(T'_k(\text{a}) + T'_k(\text{b}), (p-1)(T'_k(\text{f}) + T'_k(\text{g}))) - T'_k(R)}{(p/N)T_{\text{s}} + T'_{\text{m}}} + 1. \tag{21}$$

From (20) and (21) it can be seen that as n_{p} increases for a fixed packet size N/p, the sequence I $\rightarrow$ III $\rightarrow$ II holds if $T'_k(\text{a}) + T'_k(\text{b}) > (p-1)(T'_k(\text{f}) + T'_k(\text{g}))$. On the contrary, as N/p increases for a fixed n_{p}, the inverse sequence II $\rightarrow$ III $\rightarrow$ I holds. If $T'_k(\text{a}) + T'_k(\text{b}) < (p-1)(T'_k(\text{f}) + T'_k(\text{g}))$ then Case IV occurs instead of Case III.

The factor $(p-1)/p$ weighing the term $T_k(\text{f}) + T_k(\text{g})$ in (9) is convenient for small values of p. This factor can be easily changed without altering the main results described above. Let us consider the case where the factor is changed to $1/p$. Changing also this factor in the conditions of Cases I, II, III and IV, it is easy to see that the analysis of Cases I and IV will remain unaltered. In Case II the critic sample size will be greater than that given in (14), while in Case III the critic sample size will be greater than (14) only if $(p-2)((n_{\text{p}}+1)T_{\text{s}} - T^{p}_{\text{cp}}) > 0$.

6.2. Two-packet parallel learning algorithm

In this section the two-packet learning algorithm is presented. This version assumes that the critical path respects the scheme of Fig. 9. Since steps (a′), (g) and (d) in gating networks are out of the critical path, they can be split or located in the most convenient way. For example, step (d) (calculation of outputs) can be delayed till the end of the cycle, since outputs are needed only in the last step (h) of the root gating network (calculation of the error).

For all the expert modules: Apply each of the following steps to half the examples in the training set:

(a/2) calculate the outputs $\mu_{ij}(\vec{x})$;
(b/2) calculate the Gaussian probabilities $P_{ij}(t/\vec{x})$;
$\rightleftharpoons$ send the first half Gaussian probabilities to its parent in the tree;
(a/2) calculate the outputs $\mu_{ij}(\vec{x})$;
(b/2) calculate the Gaussian probabilities $P_{ij}(t/\vec{x})$;
$\rightleftharpoons$ receive the first half joint posterior probabilities from its parent in the tree;
$\rightleftharpoons$ send the second half Gaussian probabilities to its parent in the tree;
(f/2) update the variances σ^2_{ij};
(g/2) update the weights W;
$\rightleftharpoons$ send the first half outputs to its parent in the tree;
$\rightleftharpoons$ receive the second half joint posterior probabilities from its parent in the tree;

```
(f/2) update the variances $\sigma_{ij}^2$;
(g/2) update the weights $W$;
$\rightleftharpoons$ send the second half outputs to its parent in the tree.
```

For all the inner-level gating modules: Apply each of the following steps to half the examples in the training set, except for step (a′) which is split into 3/4 and 1/4 of the training examples:

```
(3/4 a′) calculate the prior probabilities $g_{j/i}(\vec{x})$;
$\rightleftharpoons$ receive the first half likelihoods from its two children;
(c/2) calculate the new likelihood $l_i(t|\vec{x}) = \sum_j g_{j|i}(\vec{x}) P_{ij}(t|\vec{x})$;
$\rightleftharpoons$ send the first half likelihood to its parent;
(e/2) calculate the first half conditional posterior probabilities $h_{j/i}$;
(1/4 a′) calculate the prior probabilities $g_{j/i}(\vec{x})$;
$\rightleftharpoons$ receive the first half conditional posterior probabilities from its parent;
(e′/2) calculate the first half joint posterior probabilities $h_{ij} = h_i h_{j/i}$;
$\rightleftharpoons$ send the first half joint posterior probabilities to its two children;
$\rightleftharpoons$ receive the second half likelihoods from its two children;
(c/2) calculate the new likelihood $l_i(t|\vec{x}) = \sum_j g_{j|i}(\vec{x}) P_{ij}(t|\vec{x})$;
$\rightleftharpoons$ send the second half likelihood to its parent;
(e/2) calculate the second half conditional posterior probabilities $h_{j/i}$;
(g/2) update the weights $V_1$;
$\rightleftharpoons$ receive the second half conditional posterior probabilities from its parent;
(e′/2) calculate the second half joint posterior probabilities $h_{ij} = h_i h_{j/i}$;
$\rightleftharpoons$ send the second half joint posterior probabilities to its two children;
$\rightleftharpoons$ receive the first half outputs from its two children;
(d/2) calculate the first half outputs $\mu_i(\vec{x})$;
$\rightleftharpoons$ send the first half outputs to its parent;
(g/2) update the weights $V_1$;
$\rightleftharpoons$ receive the second half outputs from its two children;
(d/2) calculate the second half outputs $\mu_i(\vec{x})$;
$\rightleftharpoons$ send the second half outputs to its parent.
```

For the gating module at the root of the tree: Apply each of the following steps to half the examples in the training set, except for step (a′) which is split into 3/4 and 1/4 and step (g) which is calculated at once:

```
(3/4 a′) calculate the prior probabilities $g_i(\vec{x})$;
$\rightleftharpoons$ receive the first half likelihoods from its two children;
(c/2) calculate the new likelihood $P(t/\vec{x})$;
(e/2) calculate the first half conditional posterior probabilities $h_i$;
$\rightleftharpoons$ send the first half conditional posterior probabilities to its two children;
(1/4 a′) calculate the prior probabilities $g_i(\vec{x})$;
$\rightleftharpoons$ receive the second half likelihoods from its two children;
(c/2) calculate the new likelihood $P(t/\vec{x})$;
```

(e/2) calculate the second half conditional posterior probabilities h_i;
$\rightleftharpoons$ send the second half conditional posterior probabilities to its two children;
(g) update the weights V_2;
$\rightleftharpoons$ receive the first half outputs from its two children;
(d/2) calculate the first half global output $\mu(\vec{x})$;
(h/2) calculate the first half error;
$\rightleftharpoons$ receive the second half outputs from its two children;
(d/2) calculate the second half global output $\mu(\vec{x})$;
(h/2) calculate the second half error and evaluate the stop criterion.

A second version of the two-packet learning algorithm was also implemented. In version 2, for all the expert modules the reception of the second half joint posterior probabilities from its parent in the tree was performed before sending the first half outputs to its parent in the tree. The rest of the modules remain the same.

6.3. Experimental time measurements

Fig. 10 illustrates an experimental time diagram measured on a two-packet pipeline algorithm learning from 8000 examples and mapped onto seven processors. A visual comparison between Figs. 6 and 10 shows that the idle time, depicted in white, has been strongly reduced in the two-packet pipeline.

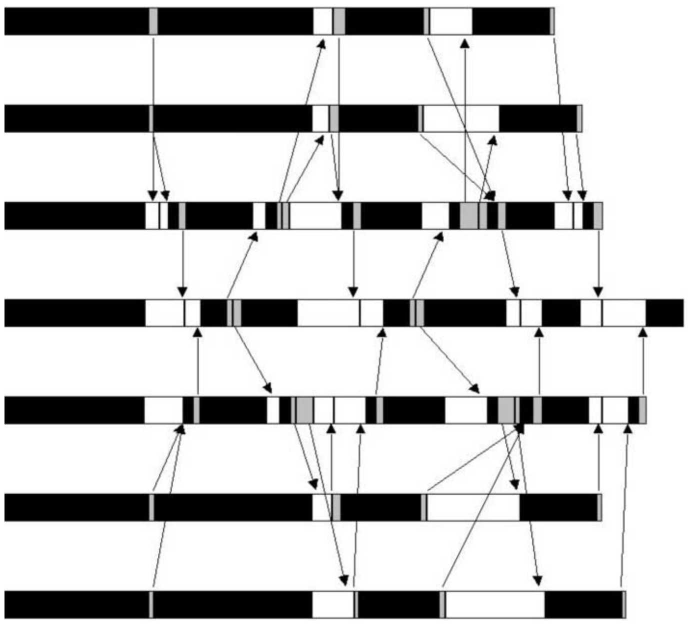

Fig. 10. Experimental time diagram for the pipelined version (linear units). Load of each processor is shown in black, message sending is depicted in grey and idle time in white.

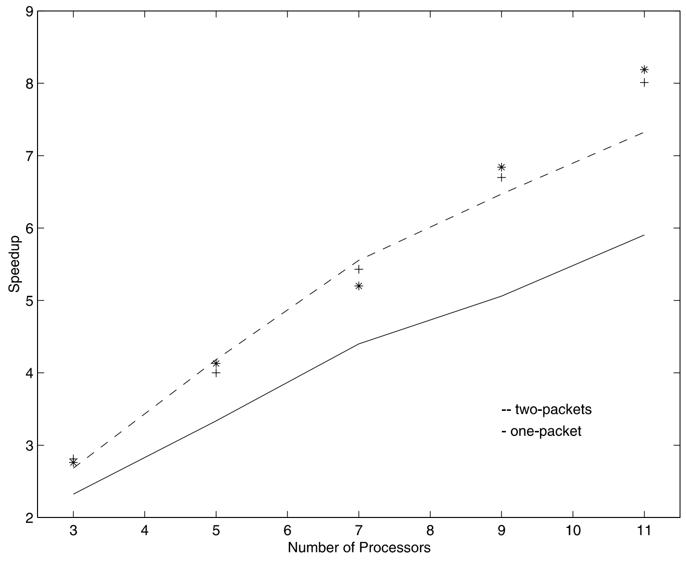

Fig. 11. Speedup as a function of n_p, with (- - -) and without (—) pipelining. Experimental results are plotted for version 1 ($*$ marks) and version 2 ($+$ marks) of the two-packet algorithm.

Fig. 11 shows experimental and theoretical speedups (for a communication time $T_m = 0.009$ s) as a function of the number of processors, for 8000 examples. Two sets of experimental results are plotted ($*$ and $+$ marks), corresponding to versions 1 and 2 of the two-packet algorithm, respectively. The lines correspond to the theoretical models for a one-packet pipeline (solid line) and for a two-packet pipeline (dashed line), with the same coefficients for the modeling obtained in Section 5.3. Note that the speedup is rather sensitive to the communication time, but all experimental points in Fig. 11 are within the range of values given in Section 5.3.

The influence of the sample size is illustrated by Fig. 12, with and without pipeline, for seven processors. Two sets of experimental results are plotted for versions 1 and 2 of the two-packet pipeline ($*$ and $+$ marks), and another set for the one-packet pipeline ($\circ$ marks). The lines correspond to the theoretical models for the one-packet pipeline (solid line) and the two-packet pipeline (dashed line), with the same coefficient values as in Section 5.3.

The theoretical models are quite accurate, since the intersection of the two curves can be explained. From (21) it can be deduced that Case II prevails for a sample size $N < 4080$, otherwise Case III predominates. From (14), the theoretical critical sample size is $N_{\text{critic}} \approx 1716$, matching well the experimental result of Fig. 12, where the two curves intersect for a sample size just over N_{critic}.

Given the experimental time measurements for the example considered, the theoretical model of Section 6.1 implies that no significant speedup enhancement (<2%) can be obtained with a three-packet algorithm with respect to a two-packet algorithm, due to the additional communication overhead. This result was confirmed

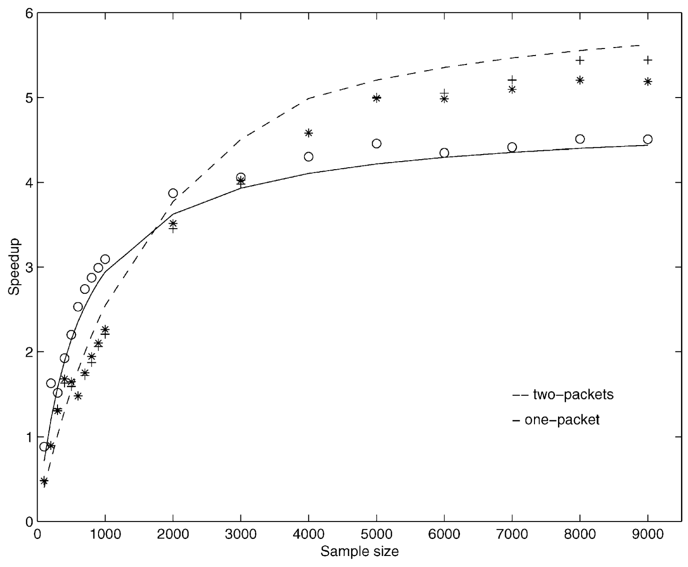

Fig. 12. Speedup as a function of the sample set size, with pipelining (- - -) and without pipelining (—). Experimental results are plotted, with pipelining ($*$ and $+$ marks, for versions 1 and 2 of the two-packet algorithm) and without pipelining ($\circ$ marks).

experimentally with a three-packet pipeline algorithm, which is not described here for the sake of space.

From (9) it can be seen that while all calculation times are reduced when p increases, the startup communication time in expression T_{cp}^{p} remains the same. Therefore, the arguments within the maximum function in (9) tend to grow with p. For single units a fraction of the calculation time becomes comparable to the term $(n_{\mathrm{p}} - 1)T_{\mathrm{s}}$. In the next section we consider the case of MLP networks instead of single units.

7. Parallel implementation of HME with MLP neural networks

All the lessons learned from the implementation of the HME model with single units can be applied to propose an accurate model and an efficient implementation for the more elaborate model of HME, with MLPs as expert and gating nodes.

When considering the HME model with MLP neural networks, there are five variables that could affect the different steps of the parallel algorithms proposed in the previous sections. These variables are:

1. The number of examples, N. This parameter affects all steps and messages. Since all examples are applied to every network, this parameter must be the same everywhere.
2. The number of inputs, n_i (dimension of the input vector). This parameter affects steps (a) and (g) in experts, and steps (a′) and (g) in gating networks, but does not

affect the size of messages. Parameter n_i could be different for each expert or gating network.

3. The number of hidden units, n_h. Without loss of generality a single hidden layer is assumed. This parameter affects steps (a) and (g) in experts, and steps (a′) and (g) in gating networks, but does not affect the size of messages. Parameter n_h could be different for each expert or gating network.
4. The number of outputs, n_o (dimension of output vector). This parameter affects all steps in experts, besides steps (d) and (h) in gating networks. Parameter n_o also affects the size of output messages. However, since the calculation and communication of outputs are not in the critical path, it is assumed that the size of messages remains fixed. By model construction parameter n_o must be the same for every expert.
5. The number of branches, n_b. It corresponds to the number of outputs in gating networks. In the models described in Sections 5 and 6, a binary tree was assumed. Parameter n_b affects all steps in gating networks, except step (h). This parameter does not affect the size of messages. For symmetry reasons, it is convenient that parameter n_b be the same at each level of the hierarchical tree architecture.

In the following section, the case of HME neural networks with regular topologies is considered, i.e. experts of the same size, as well as gating networks of the same size. This assumption entails a good load balancing over processors. The case of irregular topologies will be commented in the discussion.

7.1. Scalability analysis with MLP networks

The *isoefficiency* function can be used as a metric of scalability [16]. This function prescribes the growth rate of the problem size required to keep efficiency at a given value as n_p increases. For scalable parallel systems, efficiency can be maintained at a fixed value if the ratio $T_\mathrm{o}/T_\mathrm{seq}$ in (6) is kept at a constant value. For a desired value E of efficiency, the following relationship holds,

$$T_\mathrm{seq} = K \times T_\mathrm{o}, \tag{22}$$

where $K = E/(1-E)$ is a constant depending on the efficiency to be maintained.

Of particular interest is the case of scaling up the size of expert and gating networks, from a single unit to MLP neural networks. The size of experts is a function of the following parameters: n_ie inputs, n_he hidden units and n_o outputs. It is assumed that all experts are of the same size. Likewise, the size of gating networks is a function of the following parameters: n_ig inputs and n_hg hidden units ($n_\mathrm{b} = 2$ is fixed for a binary tree). It is assumed that all gating networks are of the same size. A large enough data set is assumed, such that $N \gg n_\mathrm{ie}, n_\mathrm{he}, n_\mathrm{ig}, n_\mathrm{hg}, n_\mathrm{o}$.

From (4), (7) and (22) the isoefficiency function yields,

$$\begin{aligned} &T^0_\mathrm{seq} + \Delta T_\mathrm{seq} = K(T_\mathrm{o} + \Delta T_\mathrm{o}), \\ &T^0_\mathrm{seq} + \Delta a_0 + \Delta a_1 n_\mathrm{p} = K(T_\mathrm{o} + \Delta k_0 + \Delta k_1 n_\mathrm{p} + \Delta k_{12} H n_\mathrm{p} + \Delta k_2 n_\mathrm{p}^2), \end{aligned} \tag{23}$$

where T_o is the overhead function and T^0_{seq} is the sequential run time when $n_{he} = n_{hg} = 0$, $n_{ie} = n_{ig} = 0$, $n_o = 1$ and $n_b = 2$, i.e. a single unit with only a bias input and a single output. ΔT_o is the increase in the overhead function and ΔT_{seq} is the increase in the sequential run time, when scaling up expert and gating networks by means of the parameters n_{ie}, n_{he}, n_o, n_{ig} and n_{hg}.

From the discussion of the previous section, we know that the four parameters (n_{ie}, n_{he}, n_{ig} and n_{hg}) changing the size of expert and gating networks only affect steps (a) and (g) in experts, and steps (a′) and (g) in gating networks (see temporal diagram of Fig. 3). Parameter n_o affects all steps in experts, besides steps (d) and (h) in gating networks. In particular, the size of messages will remain the same as with single units. Under these conditions, we have:

$$\begin{aligned}
\Delta a_0 &= \frac{1}{2}(\Delta T_k(A) - \Delta T_k(B)) + \Delta T_k(\mathrm{h}),\\
\Delta a_1 &= \frac{1}{2}(\Delta T_k(A) + \Delta T_k(B)),\\
\Delta k_0 &= -\Delta a_0,\\
\Delta k_1 &= \Delta a_0 - \Delta T_k(\mathrm{h}),\\
\Delta k_{12} &= 0,\\
\Delta k_2 &= 0.
\end{aligned} \tag{24}$$

Replacing these values in (23) and assuming that $T_k(B) = \alpha T_k(A)$ with α a real constant, we have:

$$\begin{aligned}
&T^0_{seq} + \Delta a_0 + \Delta a_1 n_p = K(T_o + \Delta a_0(n_p - 1) - \Delta T_k(\mathrm{h})n_p),\\
&\Delta a_0(1+K) + (\Delta a_1 - K(\Delta a_0 - \Delta T_k(\mathrm{h})))n_p = KT_o - T^0_{seq},\\
&(\tfrac{1}{2}(1-\alpha)(1+K) + \tfrac{1}{2}(1 - K + \alpha(1+K))n_p)\Delta T_k(A) = KT_o - T^0_{seq} - (1+K)\Delta T_k(\mathrm{h}).
\end{aligned} \tag{25}$$

Substituting $T_\alpha = \frac{1}{2}(1-\alpha)(1+K) + \frac{1}{2}(1 - K + \alpha(1+K))n_p$ in (25) and rearranging terms yields,

$$\Delta T_k(A) = \frac{KT_o - T^0_{seq} - (1+K)\Delta T_k(\mathrm{h})}{T_\alpha}. \tag{26}$$

Note that T_o is a quadratic polynomial in n_p, T^0_{seq} is linear in n_p, and T_α is also linear in n_p. From (26) it is concluded that to keep a fixed efficiency, the size of expert and gating networks should grow linearly in n_p, i.e. $\Delta T_k(A) = \mathrm{O}(n_p)$ and $\Delta T_k(B) = \mathrm{O}(n_p)$. Moreover, the models constructed in Sections 5 and 6 have assumed a critical path where the size of the gating networks is no greater than the size of experts, i.e. $\alpha \leqslant 1$.

In particular when $n_i = n_{ie} = n_{ig}$, $n_h = n_{he} = n_{hg}$ and $n_o = 1$, both expert and gating networks will grow by the same amount with n_i and n_h. This condition implies that $\alpha = 1$, $T_\alpha = n_p$ and $\Delta T_k(\mathrm{h}) = 0$. Therefore $\Delta T_o = 0$, i.e. the extra overhead time is zero. Eq. (26) is reduced to:

$$\Delta T_k(A) = \frac{KT_o - T^0_{\mathrm{seq}}}{n_p}. \tag{27}$$

If the learning algorithm is error back-propagation, outputs of $n_h + 1$ units need to be calculated in the forward phase. Likewise, in the backward phase, the error is back-propagated through the same number of units and the weights are updated. A rough estimation of the extra calculation time required by a MLP with respect to the single unit case is $n_h \times (T_\sigma + T_\delta)$, where T_σ is the calculation time of a single sigmoidal hidden unit plus the calculation time of a single linear output unit with n_h weights in the forward phase, and T_δ is the calculation time of deltas and gradients from the output unit to the hidden units and from the hidden units to the inputs, in the backward phase. Both T_σ and T_δ are functions of the number of inputs, n_i.

Replacing $\Delta T_k(A) \approx n_h(T_\sigma + T_\delta)$ in (27), and solving for the number of hidden units, yields

$$n_h \approx \frac{KT_o - T^0_{\mathrm{seq}}}{(T_\sigma + T_\delta)n_p}. \tag{28}$$

Since T_o is a quadratic polynomial in n_p and T^0_{seq} is linear in n_p, in Eq. (28), the linear term in n_p dominates the growth, i.e. $n_h = O(n_p)$ gives the overall asymptotic isoefficiency function of our parallel system, which confirms its good scalability.

7.2. Experimental time measurements

New versions of the sequential algorithm and the one-packet parallel algorithm were implemented to cope with MLPs as expert and gating networks. In the M step of the EM algorithm, the error back-propagation learning algorithm was used instead of the iteratively reweighted least squares algorithm. The computation times appearing as coefficients in expression (28) were measured for a HME network learning a mixture of two Gaussians from a data set of 8000 examples and mapped onto seven processors. Experimentally estimated parameters, for $n_h = 0$ are, in seconds:

$$k_0 = -0.0076, \quad k_1 = 0.0141, \quad k_{12} = 0.0341, \quad k_2 = 0.009,$$

$$T^0_{\mathrm{seq}} = 1.657 \quad \text{and} \quad T_\sigma + T_\delta = 0.148.$$

From the experimental value $T_\sigma + T_\delta = 0.148$, the absolute value of the relative error between the measured parallel run time and the predicted parallel run time is less than 5% in average for $n_h = 1, \ldots, 40$. Likewise, the relative error between the measured sequential run time and the predicted one is less than 5% in average for $n_h = 1, \ldots, 5$ (the sequential program runs out of memory for larger networks).

Fig. 13 shows experimental ($*$ and $\circ$ marks) and theoretical (solid lines) speedup curves parameterized in the number of hidden units, for $n_h = 0, \ldots, 10$. The experimental time measurements for $n_h = 0$ were used to estimate the theoretical model. Since the sequential program runs out of memory for $n_h \geqslant 5$, the sequential time for $n_h = 10$ was estimated as $T^0_{\mathrm{seq}} + n_p n_h(T_\sigma + T_\delta)$. In Fig. 13, fully experimental

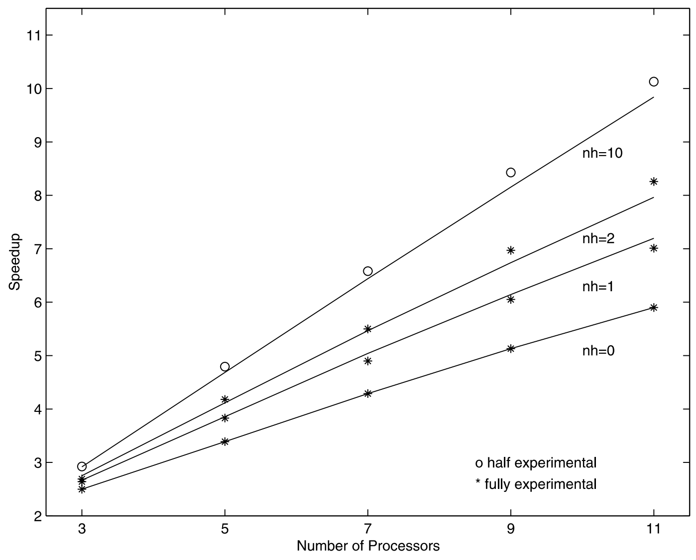

Fig. 13. Experimental ($*$ and $\circ$ marks) vs. theoretical (—) speedup as a function of the number of processors n_p and the number of hidden units n_h (for HME-MLP).

speedups are marked by stars, while half experimental speedups are marked by circles. For $n_h = 10$ near-linear speedups are reached, as can be checked in the table.

Number of processors	3	5	7	9	11
Experimental speedup	2.92	4.79	6.58	8.42	10.12

7.3. Discussion

In this work, equal sized experts have been assumed, the same for gates. Also a binary tree has been assumed. A case of particular interest is when only one gating network is used for K experts. In this case the number of branches is $n_b = K$. If the size of the gating network is smaller than the size of the experts, then the critical path will be quite similar to what was assumed in our models. On the other hand, if the size of the gating network is greater than the size of the experts, then the critical path will be given by the gating network computations. In this case, the partition of the K-ary gating network in a binary subtree of gating networks could be a good way of parallelizing the gating network computations, and our original model could be recovered.

When considering irregular topologies, for example experts and gates of arbitrary sizes, the problem of load balancing should be solved. One approach to solve this problem is graph partitioning [10]. Unfortunately, graph partitioning is a NP-complete problem, and heuristics methods must be used. The HME neural network can be seen as a directed graph, where vertices represent computations and edges represent data dependencies. The amount of work in a vertex could be represented by a weight. Similarly, a weight is associated with each edge, corresponding to the amount of communication it represents. The schematic temporal diagram of Fig. 3, gives an idea of the resulting graph for a HME network (we have to make explicit the data dependencies within experts and gates). Here computations represent calculations over the whole expert or gating network. For irregular HME networks, the graph partitioning scheme could result in the computation of part of an expert network and part of a gating network over the same processor, no longer mapping experts or gates onto different processors.

For fixed irregular topologies, the load can be estimated from the amount of connections in the network, and a static load balancing approach could be used. A more challenging problem is the case of dynamically changing HME networks. This occurs when using growing HME algorithms [37] or pruning HME algorithms [14]. Prechelt [29] has addressed the problem of developing a compiler for efficient implementation of dynamic irregular neural networks, although modular neural networks are not included.

8. Conclusion

We have shown that the modularity of HME neural architecture makes it suitable for parallelization. A simple version of the model, with single units as expert and gating networks, has been implemented both as a portable sequential program (written in C), and as a portable parallel program (also in C) using the MPI communication library (available on most parallel computers, and for PC clusters running most operating systems). The parallel algorithm has been optimized by pipelining examples into packets. A theoretical model and experimental measurements showed the limits of too small expert and gating neural networks. It has been shown that for regular topologies the proposed parallel algorithms are highly scalable when the size of the expert and gating networks grows from single units to MLPs. Since the speedup depends on the size of expert and gating networks, it is easier to reach higher, near-linear speedups, with HME-MLP.

This work can be considered as a case study in the parallelization of HME networks trained by the EM algorithm. The models presented here apply straightforwardly to different optimizations of the weights in the expert and gating networks in the M-step, such as several second order methods [1,37,38,40]. Some of the concepts presented here may be applied to Bayesian methods for training mixtures of experts, that combine the standard EM learning algorithm with re-estimation of hyper-parameters of priors on gate and expert parameters [37,38].

Furthermore, parallel machines provide the necessary amount of memory required by large real-world applications. In conclusion, our parallel algorithm for training a hierarchical mixture of neural experts has been shown to be efficient and capable of overcoming the limitations of the sequential algorithm for processing real applications on complex problems and large data bases. They confirm the interest of parallelizing modular neural networks on the basis of modular parallelism, i.e. parallel execution of network modules.

Acknowledgements

Supported by INCO-DC under the PARALIN European Project and Conicyt-Chile under grants Fondef 1022 and Fondecyt 1980909. The French participation to this work has been initiated at LIP laboratory, URA CNRS 1398, until 1998, Ecole Normale Supérieure de Lyon, France.

References

[1] K. Chen, L. Xu, H. Chi, Improved learning algorithms for mixture of experts in multiclass classification, Neural Networks 12 (1999) 1229–1252.

[2] M. Cosnard, J.-C. Mignot, H. Paugam-Moisy, Implementations of multilayer neural networks on parallel architectures, in: IEE Proceedings of 2nd International Special Seminar on Parallel Digital Processors, vol. 334, 1991, pp. 43–47.

[3] V. Demian, J.-C. Mignot, Implementation of the self-organizing feature map on parallel computers, Computers and Artificial Intelligence 14 (1) (1996) 63–80.

[4] A.P. Dempster, N.M. Laird, D.B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, Journal of the Royal Statistical Society B 39 (1) (1977) 1–38.

[5] P.A. Estévez, R. Nakano, Hierarchical mixture of experts and max–min propagation neural networks, in: Proceedings of ICNN'95, IEEE International Conference on Neural Networks, Perth, Australia, vol. 1, 1995, pp. 651–656.

[6] P.A. Estévez, H. Paugam-Moisy, D. Puzenat, M. Ugarte, Modular parallel implementation for HME neural networks, in: Proceedings of PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, vol. 4, 1998, pp. 1365–1372.

[7] J. Ghosh, K. Hwang, Mapping neural networks onto message-passing multicomputers, Journal of Parallel and Distributed Computing 6 (2) (1989) 291–330.

[8] J. Gregor, D.A. Huff, A computational study of the focus-of-attention EM–ML algorithm for PET reconstruction, Parallel Computing 24 (1998) 1481–1497.

[9] B. Hendrickson, R. Leland, The Chaco User's Guide, version 2.0, Technical Report SAND95-2344, Sandia National Labs., Alburquerque, NM, 1995.

[10] B. Hendrickson, R. Leland, Graph partitioning models for parallel computing, Parallel Computing 26 (2000) 1519–1534.

[11] H. Hopp, L. Prechelt, CuPit-2: A Portable parallel programming language for artificial neural networks, in: A. Sydow (Ed.), Proceedings of the 15th IMACS World Congress Scientific Computation Modeling and Applied Math., vol. 6, Wissenschaft and Technik Verlag, Berlin, 1997, pp. 493–498.

[12] R.A. Jacobs, M.I. Jordan, A.G. Barto, Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks, Cognitive Science 15 (1991) 219–250.

[13] R.A. Jacobs, M.I. Jordan, S.E. Nowlan, G.E. Hinton, Adaptive mixture of experts, Neural Computation 3 (1991) 79–87.

[14] R.A. Jacobs, F. Peng, M.A. Tanner, A Bayesian approach to model selection in hierarchical mixture of experts architectures, Neural Networks 10 (1997) 231–241.

[15] M.I. Jordan, R.A. Jacobs, Hierarchical mixture of experts and the EM algorithm, Neural Computation 6 (1994) 181–214.

[16] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing, Benjamin/Cummings, CA, 1994.

[17] V. Kumar, S. Shekkar, M.B. Amin, A scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures, IEEE Transactions on Parallel Distributed Systems 5 (1994) 1073–1090.

[18] M. Mangeas, A.S. Weigend, C. Muller, Forecasting electricity demand using nonlinear mixture of experts, in: Proceedings of WCNN'95, World Conference on Neural Networks, vol. 2, 1995, pp. 48–53.

[19] J. Mattes, D. Trystram, J. Demongeot, Parallel image processing using neural networks: Applications in contrast enhancement of medical images, Parallel Processing Letters 8 (1) (1998) 63–76.
[20] M. Misra, Parallel Environments for Implementing Neural Networks, Neural Comp. Survey. Available from http://www.icsi.berkeley.edu/~jagota/NCS, 1998, pp. 60–113.
[21] J.M.J. Murre, Transputers and neural networks: An analysis of implementation constraints and performance, IEEE Transactions on Neural Networks 4 (2) (1993) 284–292.
[22] H. Paugam-Moisy, Optimal speedup conditions for a parallel back-propagation algorithm, in: Proceedings of CONPAR'92-VAPP V, Lecture Notes in Computer Science, vol. 682, Springer-Verlag, 1992, pp. 719–724.
[23] H. Paugam-Moisy, Parallel neural computing based on network duplicating, in: I. Pitas (Ed.), Parallel Algorithms for Digital Image Processing Computer Vision and Neural Networks, Wiley, New York, 1993, pp. 305–340.
[24] H. Paugam-Moisy, Multiprocessor simulation of neural networks, in: M. Arbib (Ed.), The Handbook of Brain Theory and Neural Networks, MIT Press, Cambridge, MA, 1995, pp. 605–608.
[25] H. Paugam-Moisy, Neural networks: From massively parallel processing towards modular distributed processing, in: Proceedings of the VIII Congreso Latino-americano de Control Automatico, Vina del Mar, Chile, vol. 1, 1998, pp. 31–36.
[26] H. Paugam-Moisy, A. Pétrowski, Parallel neural computation based on algebraic partitioning, in: I. Pitas (Ed.), Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks, Wiley, New York, 1993, pp. 259–304.
[27] A. Pétrowski, G. Dreyfus, C. Girault, Performance analysis of a pipelined back-propagation parallel algorithm, IEEE Transactions on Neural Networks 4 (1993) 970–981.
[28] A. Pétrowski, Algorithmes Parallèles de Rétro-propagation des Erreurs pour les Réseaux de Neurones, Ph.D. Thesis, MASI, Université P. et M. Curie, Paris, France, 1993.
[29] L. Prechelt, Exploiting domain-specific properties: Compiling parallel dynamic neural network algorithms into efficient code, IEEE Transactions on Parallel and Distributed Systems 10 (11) (1999) 1105–1117.
[30] L. Prylli, CAPNX, un environement NX, PVM et MPI multi-utilisateurs sur MCS Capitan, Research report 95-48, ENS-Lyon, France, 1995.
[31] D. Puzenat, Parallélisme et Modularité des Modèles Connexionnistes, Ph.D. Thesis, LIP, ENS Lyon, France, 1997.
[32] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation, in: Parallel Distributed Processing, vol. 1, MIT Press, Cambridge, MA, 1986, pp. 318–362.
[33] P. Saratchandran, N. Sundarajan, S.K. Foo, Parallel Implementations of Backpropagation Neural Networks on Transputers, World Scientific, Singapore, 1996.
[34] J.J.E. So, T.J. Downar, R. Janardhan, H.J. Siegel, Mapping conjugate gradient algorithms for neutron diffusion: applications onto SIMD, MIMD, and mixed-mode machines, International Journal of Parallel Programming 26 (2) (1998) 183–207.
[35] V. Sudhakar, C. Siva Ram Murthy, Efficient mapping of backpropagation algorithm onto a network of workstations, IEEE Transactions on Systems, Man and Cybernetics, Part B 28 (6) (1998) 841–848.
[36] S. Wang, Reducing the communication cost in simulating layered neural networks on a hypercube machine, in: Proc. of Parallel Computing'89, Elsevier, Amsterdam, 1989, pp. 375–380.
[37] S.R. Waterhouse, Classification and Regression using Mixture of Experts, Ph.D. Thesis, Department of Engineering, University of Cambridge, UK, 1997.
[38] S.R. Waterhouse, A.J. Robinson, Classification using hierarchical mixture of experts, in: Proc. IEEE Work. on Neural Networks for Signal Processing, 1994, pp. 177–186.
[39] S.R. Waterhouse, A.J. Robinson, Nonlinear prediction of acoustic vectors using hierarchical mixture of experts, in: Proceedings of NIPS'94, Advances in Neural Information Processing Systems, MIT Press, Cambridge, MA, 1994, pp. 835–842.
[40] A.S. Weigend, M. Mangeas, A.N. Srivastava, Nonlinear gated experts for time series: Discovering regimes and avoiding overfitting, International Journal of Neural Systems 6 (4) (1995) 373–399.