Workshop 11

High Performance Computing and Application

Parallel Implementation of RBF Neural Networks

V. Demian[1] and F. Desprez[2] and H. Paugam-Moisy[2] and M. Pourzandi[3]

[1] LaBRI, Université Bordeaux 1, F-33405 Talence cedex, France
[2] LIP, Ecole Normale Supérieure de Lyon, F-69364 Lyon cedex 07, France
[3] LIFL, Université Lille 1, F-59655 Villeneuve d'Ascq cedex, France

Abstract. This report presents several parallel implementations, on a MIMD machine, of a learning algorithm called OLS (Orthogonal Least Squares) for RBF (Radial Basis Function) neural networks. The sequential version is first described, and a straightforward parallel version is proposed. Two variants are developed, one of them reducing the complexity of the algorithm, and the other one improving the load balancing. An alternative is proposed for the storage of initial or intermediate data on local memory and discussed, according to the size of the application.

1 Introduction

A method based on using Radial Basis Functions (RBF) has been first introduced in 1988 by Broomhead and Lowe for resolving applications by neural networks [1]. Now RBF neural networks can be considered as an alternative to multi-layer perceptrons (MLP). They principally differ from MLP by the local representations processed by their internal units. RBF neural networks are a powerful tool for computing complex functions since it has been proved that they are universal approximators [9, 10], just as multi-layer perceptrons be [3, 6]. RBF neural networks achieve good performance with rather fast learning algorithms which make them particularly interesting to be used in a wide range of computational resource consuming applications, such as pattern recognition, signal processing, function approximation, classification, etc [11, 12].

In 1989, Moody and Darken proposed the first learning algorithms for this connectionist model [8]. Beside a completely supervised method based on gradient descent, they proposed a two-phases algorithm based on self-organization for the hidden layer and then on supervised learning for the output layer. In order to overcome the difficulty of choosing the number and value of hidden units, a priori, in 1991 Chen and al. [2] proposed a new learning rule for RBF neural networks which they called the OLS learning algorithm. An arbitrary choice of the initial positions of centers often yields a bad performance (local minima) or a too large size for the network. Contrary to classical algorithms, the OLS learning algorithm does not modify the positions of initial centers but selects them among the initial set of examples.

The sequential version of the OLS algorithm is first summarized, and a straightforward parallel version is proposed. Afterwards, two variants are developed for the parallel version and their convenience is discussed.

2 Description of the OLS learning algorithm

2.1 RBF neural network: architecture and computation

An RBF neural network is composed of n input units and one hidden layer of radial basis function units. Its scalar output computes the following function

$$f(\boldsymbol{x}) = \theta_0 + \sum_{i=1}^{h} \theta_i \Phi(\| \boldsymbol{x} - \boldsymbol{c_i} \|), \tag{1}$$

where $\boldsymbol{x} \in \mathcal{R}^n$ is an input vector, $\theta_{i,0 \leq i \leq h}$ are the weights, $\Phi(\cdot)$ is a function $\mathcal{R}^+ \rightarrow \mathcal{R}$, $\|\cdot\|$ is the Euclidean norm, $\boldsymbol{c}_{i,1 \leq i \leq h}$ are the RBF centers and h is the number of centers. The function Φ can be for instance a Gaussian or a thin_plate_spline function. The architecture described above, with only one scalar output, can be easily extended to networks with a vector for output, by replicating the last layer (with new weights) towards multiple scalar outputs [2]. Learning is usually divided in two phases: first determining the first layer and the centers $\boldsymbol{c}_i$, and second adjusting the output weights θ_i by a Least Square (LS) algorithm.

2.2 Orthogonal Least Squares (OLS) learning algorithm

Principle In usual learning algorithms for RBF networks, the number h of initial centers is fixed arbitrarily and these centers are randomly chosen among the database. In their learning algorithm, Chen et al. randomly choose a set of M potential centers among the example database and construct the hidden layer by an iterative procedure, adding one new center at each step. The RBF network is seen as a special case of the linear regression model and rewritten as follows (see [2] and [4] for details)

$$d(t) = \sum_{i=1}^{M} p_i(t)\theta_i + \epsilon(t), \tag{2}$$

where $d(t)$ is the desired output for example $\boldsymbol{x}(t)$, θ_i are the parameters and the outputs of hidden units $p_i(t) = \Phi(\| \boldsymbol{x}(t) - \boldsymbol{c}_i \|)$ are the regressors. Since $d(t) = f(\boldsymbol{x}(t)) + \epsilon(t)$, the term $\epsilon(t)$ represents the error signal. Now consider every example $\boldsymbol{x}(t)$ of a database of size N. Then equation (2) yields to the following matrix form

$$\boldsymbol{d} = \boldsymbol{P\Theta} + \boldsymbol{E} \tag{3}$$

where $\boldsymbol{P}$ is a $N \times M$ matrix. The OLS algorithm involves a transformation of $\boldsymbol{P}$ into a set of orthogonal basis vectors, in order to evaluate the individual contribution to the desired output from each basis vector. Matrix $\boldsymbol{P}$ is decomposed into $\boldsymbol{P} = \boldsymbol{WA}$ where $\boldsymbol{A}$ is a $M \times M$ triangular matrix and $\boldsymbol{W}$ is a $N \times M$ matrix with orthogonal columns.

A new center *imax* is selected in order to maximize the explained variance of the desired output. Each selected center is suppressed from the set $\mathcal{S}$ of potential centers, the size of which is reduced to $M - k$ after k iterations. The stop criterion can be $M_{stop} = M$ or governed by a threshold on the error.

Sequential algorithm and data structure The initial step of the iteration series is reduced to the selection of $\boldsymbol{w}_1^{imax} = \boldsymbol{p}_{imax}$ for which the error reduction ratio is maximum, without orthogonalization process. Further steps are described by the following algorithm

```
repeat
    for every potential center in S
        compute p^(center)      % column vector of P, for this center
        orthogonalize: w^(center) <-- Gram-Schmidt (p,w_1,...,w_(k-1))
        compute the error reduction for this center
    determine the center which induces the maximum error reduction
    select this center: w_k <-- w^(imax)
    suppress this center from S
    k <-- k+1
until stop_criterion or S = empty_set
```

3 Parallel implementations

In order to construct RBF networks with the best learning performance, the OLS algorithm should be executed with $M = N$, i.e. the set of potential centers being the whole database. However, such a version of the algorithm cannot be run on a sequential machine due to its high cost in CPU time and memory capacity. Indeed, under this assumption, the time complexity of the algorithm is $O(N^4)$ and the required memory is $O(N^2)$, which would not be realistic for a real-world application with several thousands of examples. Then parallel computing is useful for solving a larger class of problems with RBF networks.

In this section, first we describe a straightforward parallel version of the OLS learning algorithm, second we propose an improvement based on algorithmic considerations. In both cases, we discuss the opportunity of storing different data structures, according to the size of the problem. Finally, we propose a last improvement based on a dynamic load balancing algorithm.

3.1 Straightforward parallel algorithm

From the description of the sequential algorithm, it comes that the simplest way to allocate the computation and data to several processors consists in splitting the computation of matrix $\boldsymbol{P}$ into blocks composed of identical numbers of columns. Such a data allocation involves communications for determining the global maximum of the error reduction and for updating the copies of matrix $\boldsymbol{W}$. Let *nb_proc* be the number of processors. The local portion of $\boldsymbol{P}$ is reduced to size $N \times M/nb\_proc$. Hence the computational time on each processor is reduced, especially when the number of processors is large.

3.2 Improvement of the parallel implementation

In this section, we explain how to decrease the computational complexity by an order of magnitude, by storing a part of intermediate results in local memory, on

every processor. Therefore, we must detail the Gram-Schmidt procedure: $w^{center} \leftarrow$ Gram-Schmidt $(p, w_1 \cdots w_{k-1})$. For iteration k, this procedure can be developed as follows, where a_j^m are the components of matrix A

$$\text{for } j = 1 \text{ to } k-1 \text{ do } a_j^{center} = \frac{w_j^T p}{w_j^T w_j}$$

$$w_k^{center} = p - \sum_{j=1}^{k-1} a_j^{center} w_j$$

Hence, in the next iteration, only the new values a_k^{center}, for $center = 1$ to $nb\_remaining\_examples$, have to be computed and the last computation line can be rewritten

$$w_{k+1}^{center} = w_k^{center} - a_k^{center} w_k$$

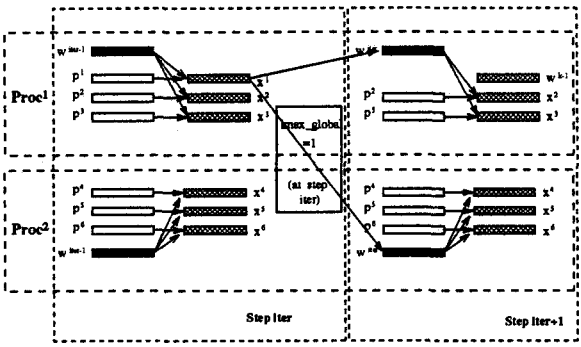

Fig. 1. Storing intermediate results in matrix X.

Thus the computational complexity can be reduced if the value x^{center} of the orthogonal image w_k^{center} of every p^{center} is stored in memory. This remark holds for both the sequential algorithm and the parallel one. Matrix X is composed of the x^{center} vectors as columns. At the k^{th} iteration, the size of matrix X is $(M-k) \times N$. In the previous version of the parallel algorithm, one copy of matrix W had to be stored on each processor, which is no longer necessary if matrix X is stored. It is sufficient to store the last w_{k-1} column of matrix W, as shown on Figure 1. Furthermore, only the part of matrix X corresponding to the local block of P has to be stored on each processor, which is smaller than the matrix W, since the number k of iterations is sufficiently large.

In this version of the parallel algorithm, the code of the inner loop implemented on each processor is modified as follows

$$x_{center\_proc} \leftarrow \text{Gram-Schmidt } (x_{center\_proc}, w_{k-1})$$

Since the processor owning the winner broadcasts x_{imax_global}, the other processors receive it as w_k.

The computational complexity of the original sequential algorithm is $O(N^4)$, due to the inner loop of the Gram-Schmidt procedure. In the new version, the complexity of this loop is reduced to $O(N^3)$ which is also the complexity of the computation of the p vectors. Hence the complexity of the whole algorithm has been reduced to $O(N^3)$. Furthermore, the required memory size on each processor has been reduced too.

3.3 Storage of initial data

The first idea consists in storing the whole example database on each processor, which allows to compute all the p^{center} vectors without communication. The total size of this database is $N \times dim$, where dim is the dimension of the input space. However the computation of the p vectors is the same at each iteration. This repetitive computation can be avoided if the memory space is sufficiently large to store the matrix P, the size of which is $N \times M$ in the sequential algorithm and $N \times M/nb\_proc$ on each processor in parallel algorithms.

Usually M is much larger than dim and the storage of the database is the good solution for the sequential algorithm. Nevertheless, the opportunity of storing the matrix P (by blocks) rather than the database is worth being discussed for parallel algorithms. The following condition determines which version to use.

$$\text{If} \quad N \times \frac{M}{nb\_proc} < N \times dim \quad \text{i.e.} \quad M < dim \times nb\_proc$$

then the version with a block of the matrix P in local memory, which will be called version I, has to be preferred to the version without storage of matrix P, further denoted by version II. This situation occurs either for a large value of dim or for a small number of potential centers. Figure 2 presents the variations of the memory size, for the improved parallel algorithm, according to the values of N and dim, under the assumption $M = N$. For sufficiently large dim and small M, the curve of version I is below the curve of version II, which means that the storage of P is the best solution in these conditions.

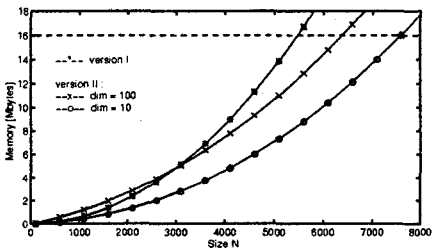

Fig. 2. Memory size of local data for versions I and II of the improved parallel algorithm, for $nb\_proc = 30$ processors. For version II, we give the curves for $dim = 10$ and $dim = 100$.

3.4 Dynamic load balancing

Dynamic load balancing problem As presented in the previous sections, selected centers are suppressed from the set of potential centers according to the maximization of the variance of the desired output. After k iterations, k examples have been suppressed from the initial set S. This suppression can occur anywhere on the processors and thus it is impossible to predict the number of remaining examples on one processor. In the best case, centers are chosen alternatively on each processor and the amount of work stays the same. In the worst case, at iteration k, all the $k-1$ previous centers have been chosen on the first processors and a load inbalance appears in the amount of work for all the processors. Since the all-to-all communication (for determining the global maximum) introduces a synchronization point

at each iteration, the execution time is limited by the one of the slowest processor. The goal of the load balancing is to keep the number of examples nearly identical on the processors at every iteration. Therefore, examples are moved between processors during the execution of the algorithm if an inbalance occurs. This kind of load balancing is also used in image processing (for example, see [7]).

Implementation within the algorithm At each iteration, the logical number of the processor holding the chosen center w_{iter} is sent during the broadcast. Thus, each processor knows the number of remaining examples on every processor by keeping a table up-to-date. Then, each processor computes the optimal number of examples in each processor ($nb\_opt = \frac{nb\_total\_exemples}{nb\_proc}$). Processors owning too many examples can send some of them to other processors. Because of the communication cost of this load balancing, a tradeoff has to be found. The load balancing procedure is called when $nb\_opt - nb_i > threshold$, where *threshold* is given by the user.

The communication cost of this algorithm is at most equal to the communication time for sending *threshold* examples from one processor to an other.

Experimental results Our target machine is an Intel Paragon with 30 nodes. Briefly, it is a distributed memory machine in which i860 processors are connected on a grid. A complete description of its architecture can be found in [5]. To obtain good performances and a good scalability, libraries are used for communications (BLACS) and for computations (level 1 BLAS).

Figure 3 presents the different execution times for both versions I and II. The execution time is lower for version I than for version II. However, as this last version needs less memory, it allows to run the algorithms on twice the number of examples. There is again a tradeoff between the size of the base and the execution time.

Figure 4 shows the speed-ups with and without load balancing as a function of the number of processors for a base of 1000 examples. Using load-balancing, we obtain a quasi-linear speed-up.

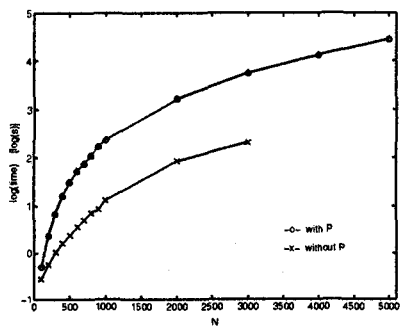

Fig. 3. Execution times for versions I and II as a function of the number of examples (30 processors, $dim = 10$).

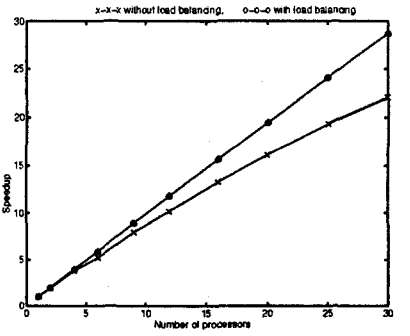

Fig. 4. Speedup with and without load balancing (1000 examples, $dim = 10$, threshold $= 1$).

Figure 5 show the speed-up curves for the versions with and without load balancing for the random and worst cases. By "random", we mean the case where the example selected as a new center lies randomly on each processor. The "worst case"

means that, at the k^{th} iteration, the first $k-1$ p_i have already been chosen (this is clearly an instance of the most unbalanced case). We obtain a 20% gain for the random case and about 44% in the worst case.

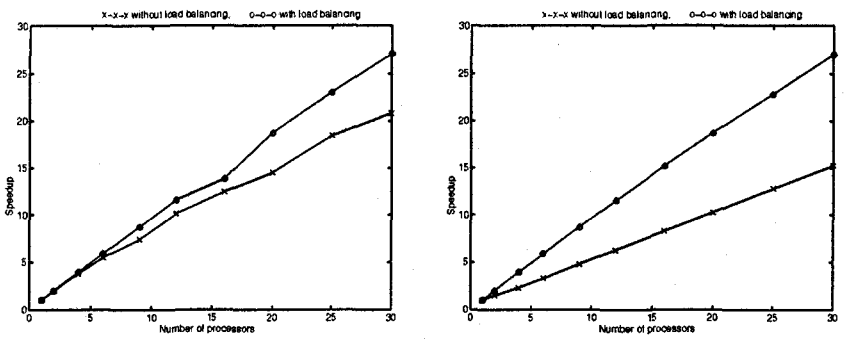

Fig. 5. SpeedUps with and without the dynamic load balancing for the random and worst cases as a function of the number of processors (500 examples, $dim = 10$, threshold $= 1$).

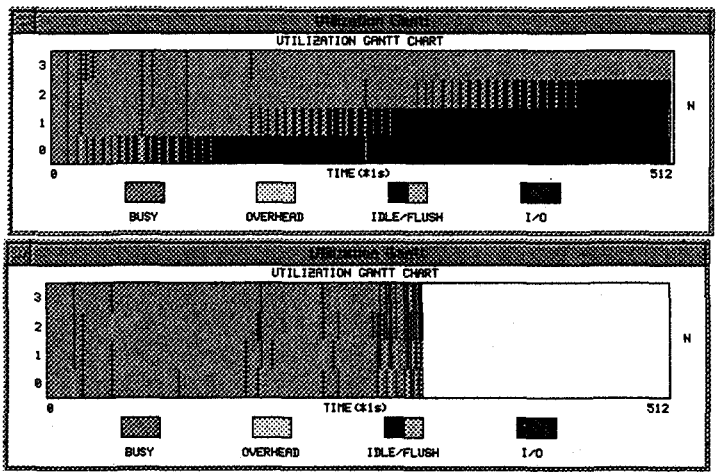

Fig. 6. Paragraph Gantt charts of the solution with or without dynamic load balancing for the worst case (500 examples, $dim = 10$, 4 processors).

Figures 6 show the Paragraph Gantt charts of the execution of the algorithm for 500 examples with $dim = 10$ on 4 processors with or without load balancing (in light gray when processors are busy and in dark when they are idle). The improvements of the execution time is evident. Execution time is almost divided by two and the processors are never idle when load balancing is implemented.

4 Conclusion

In this paper, we proposed several parallel algorithms for learning RBF neural networks with the OLS algorithm starting from the sequential algorithm of Chen et al. We presented and compared the performances of several implementations on an MIMD machine, the Intel Paragon.

We proposed an improvement of the original algorithm, which reduces the complexity from $O(N^4)$ to $O(N^3)$. This variant is specially attractive for the parallel algorithm, since the local memory required by the local blocks of matrix X is generally smaller than the memory size necessary for storing the whole matrix W.

We showed that storing intermediate results (matrix P) instead of the initial database can reduce the computation time without increasing the local memory size, under assumptions. We compared the different versions and pointed out the advantages and drawbacks of each one, according to the size of the application.

Althought all the parallel versions show good performances we enhance the substantial gains observed in implementing a dynamic load balancing algorithm. Nevertheless, our load balancing algorithm is not yet optimal and it could be improved in further works.

References

1. D. S. Broomhead and D. Lowe. Multivariable Functional Interpolation and Adaptive Networks. *Complex Systems*, 2:321–355, 1988.
2. S. Chen, C. F. N. Cowan, and P. M. Grant. Orthogonal Least Squares Learning Alg. for Radial Basis Function Networks. *IEEE Trans. on NN*, 2(2):302–309, March 1991.
3. G. Cybenko. Approximation by Superposition of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.
4. V. Demian, F. Desprez, H. Paugam-Moisy, and M. Pourzandi. Parallel Implementation of RBF Neural Networks. Technical Report 96-11, LIP ENS Lyon, 1996.
5. R. Esser and R. Knecht. Intel Paragon XP/S - Architecture and Software Environment. Technical Report KFA-ZAM-IB-9305, Central Institute for Applied Mathematics, Research Center Julich (KFA), April 1993.
6. K. Hornik, M. Stinchcombe, and H. White. Universal Approximation of an Unknown Mapping and its Derivatives Using Multilayer Feedforward Networks. *Neural Networks*, 3(5):551–560, 1990.
7. Serge Miguet and Yves Robert. Elastic Load Balancing for Image Processing Plgorithms. In H. P. Zima, editor, *Parallel Computation*, Lect. Notes in Comp. Sci., pages 438–451, Salzburg, Austria, September 1991. 1st Int. ACPC Conf., Springer Verlag.
8. J. Moody and C. J. Darken. Fast Learning in Networks of Locally Tuned Processing Units. *Neural Computation*, 1(2):281–294, 1989.
9. J. Park and I. W. Sandberg. Universal Approximation Using Radial-Basis-Function Networks. *Neural Computation*, 3(2):246–257, 1991.
10. J. Park and I. W. Sandberg. Approximation and Radial Basis Function Networks. *Neural Computation*, 5(2):305–316, 1993.
11. S. Renals and R. Rohwer. Phoneme Classification Experiments Using Radial Basis Function. *Proc. of Int. Joint Conf. on Neural Net.*, 1:461–467, 1989.
12. G. Verckovnik, C. R. Carter, and S. Haykin. Radial Basis Function Classification of Impulse Radar Waveforms. *Proc. of Int. Joint Conf. on Neural Net.*, 1:45–50, 1990.