PVM–based Training of Large Neural Architectures

V.P. Plagianakos[1,3], G.D. Magoulas[2,3], N.K. Nousis[1,3], and M.N. Vrahatis[1,3]

(1)Department of Mathematics, University of Patras,
GR-261.10 Patras, Greece.
e–mail: {vpp,nousis,vrahatis}@math.upatras.gr

(2)Department of Information Systems and Computing, Brunel University,
Uxbridge UB8 3PH, UK.
e–mail: George.Magoulas@brunel.ac.uk

(3)University of Patras Artificial Intelligence Research Center–UPAIRC.

Abstract

In this paper a methodology for parallelizing neural network training algorithms is described, based on the parallel evaluation of the error function and gradient using the Parallel Virtual Machine (PVM). PVM is an integrated set of software tools and libraries that emulates a general–purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architectures. The proposed methodology has large granularity and low synchronization, and has been implemented and tested. Our results indicate that the relatively easy setup of the PVM (using existing workstations), and parallelization of the training algorithms results in considerable speedups especially when large network architectures and training vectors are used.

1 Introduction

The general use of a Multi–Layer Perceptron (MLP) consists of a training phase followed by the classification phase. The training phase involves an optimization procedure and can be very time consuming (especially when large network topologies and/or training data are used), since a feasible minimum in the weight space is sought. On the other hand the classification of an unknown test vector is extremely fast, since it requires only the propagation of the test vector through the neural network.

In the Back-Propagation (BP) class of algorithms the optimization procedure used consists of the following steps:

1. Presentation to the MLP of all the training examples and computation of the activation of the network.

2. Computation of the error function based on the activation (usually the sum of squared differences between the actual and the desired output).

3. Computation of the gradients at a point in the weight space.

4. Adaptation of the weights of the MLP according to the training algorithm used.

The Steps 1, 2 and 3 can be easily performed in parallel, if the training set is partitioned across multiple processors. On the other hand, Step 4 is better to be performed at only one processor, after the partially evaluated error function and gradient values are sent to it and accumulated.

Formally, let us consider an MLP whose l-th layer contains N_l neurons, $l = 1, \ldots, M$. The network is based on the following equations:

$$net_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1}, \qquad y_j^l = f(net_j^l),$$

where $w_{ij}^{l-1,l}$ is the weights from the i-th neuron at the $(l-1)$ layer to the j-th neuron at the l-th layer, y_j^l is the output of the j-th neuron that belongs to the l-th layer, and $f(net_j^l)$ is the j-th's neuron activation function. The training process can be realized by minimizing the error function E defined by:

$$E = \frac{1}{2} \sum_{p=1}^{P} \sum_{j=1}^{N_M} \left(y_{j,p}^M - t_{j,p} \right)^2 = \sum_{p=1}^{P} E_p, \qquad (1)$$

where $\left(y_{j,p}^M - t_{j,p} \right)^2$ is the squared difference between the actual output value at the j-th output layer neuron for the pattern p and the target output value, and p is an index over input-output pairs. The function E also provides the error surface over the weight space.

To simplify the formulation of the above equations, let us use a unified notation for the weights. Thus, for an MLP with n weights, let w be a column weight vector with components $w_1, w_2, \ldots, w_n$, defined on the n–dimensional Euclidean space $\mathbb{R}^n$, and $w^\star$ be an optimal weight vector with components $w_1^\star, w_2^\star, \ldots, w_n^\star$. Also, let E be the batch error measure defined as the sum of squared differences error function over the entire training set, and $\nabla E(w)$ be the gradient vector of the error function E at w.

The minimization of function E corresponds to the weight update by epoch and, requires a sequence of weight vectors $\{w^k\}_{k=0}^{\infty}$, where k indicates epochs. Successful training implies that $\{w^k\}_{k=0}^{\infty}$ converges to the point $w^\star$ that minimizes E. The widely used gradient–based training algorithm BP, minimizes the error function using the following steepest descent method with constant, heuristically chosen, learning rate η:

$$w^{k+1} = w^k - \eta \nabla E(w^k). \qquad (2)$$

The remaining of the paper is organized as follows: the Parallel Virtual Machine (PVM) is briefly described in Section 2, while the proposed methodology is explained in Section 3. In Section 4, the experimental results are exhibited. Section 5 concludes the paper.

2 The Parallel Virtual Machine

Although MLPs have been widely used in many application areas, real world problems demand an increasing amount of computational resources. However, not many neural network researchers have access to a high performance parallel machine [2]. On the other hand, most of the researchers have access to networked workstations that can be used as a Parallel Virtual Machine–PVM [3]. To this end, we have built a PVM, in order to implement and test the parallel versions of several learning algorithms.

PVM is a de facto standard message passing interface. It is an integrated set of software tools and libraries that emulates a general–purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architectures. PVM is designed to link computing resources and provide users with a parallel platform for running their computer applications, irrespective of the number of different computer architectures they use and where those computers are located. It is capable of harnessing the combined resources of typically heterogeneous networked computing platforms to deliver high levels of performance and functionality.

The PVM system uses the message–passing model to allow programmers to exploit the distributed computing across a wide variety of computer architectures, including Massively Parallel Processors (MPPs). Its key concept is that it makes a collection of computers to appear as one large *virtual* machine, hence its name [3].

Parallel processing, i.e. the method of having many small tasks solve one large problem, has emerged as a key enabling technology in modern computing. The past several years have witnessed an ever–increasing acceptance and adoption of parallel processing, both for high–performance scientific computing and for more "general–purpose" applications, was a result of the demand for higher performance, lower cost, and sustained productivity. The acceptance has been facilitated by two major developments: (a) massively parallel processors), and (b) the widespread use of distributed computing. MPPs are now the most powerful computers in the world. These machines combine a few hundred to a few thousand CPUs in a single large cabinet connected to hundreds of gigabytes of memory and offer enormous computational power.

On the other hand, distributed computing is a process whereby a set of computers connected by a network are used collectively to solve a single problem. The combined computational resources of several general–purpose workstations, interconnected with a high–speed local area network, may exceed the power of a single high performance computer.

The most important factor in distributed computing is the high cost of the hardware. Large MPPs typically cost more than \$10 million. In contrast, users see very little cost in running their problems on a local set of existing computers. Even building a PVM using dedicated computers has a reduced cost. The cost of a 15–node system is less than \$10,000 due to the use of Beowulf–style nodes [1, 12]. It must be noted that when using Beowulf nodes only the master node needs hard disk, video display, monitor and keyboard. The cost of materials for the described PVM topology can be found in Appendix 1.

3 The Proposed Methodology

Here, we have constructed a parallel procedure that uses the well known master–slave computational model [3]. Specifically, 15 slaves and one master node were connected (using a 100 Mbps Ethernet switch) to create the PVM. The PVM consists of a daemon process running in the background of each node that forms the virtual machine. The daemons are responsible for spawning the tasks (program execution) on the host machines, the synchronization, and the communication between the tasks. Our implementation is based on the partitioning of the training set across multiple processors on the slave nodes. This results in the parallel evaluation of the error function and gradient of the MLP. Below, we describe the algorithms for the master and slave nodes.

3.1 The Algorithm of the Master Node

At the beginning of the procedure, the master node adds the slave nodes to the PVM, spawns the slave tasks, partitions the initial training set into subsets (one for each slave node), and sends the network architecture and the partitions of the training set to the slaves. Then, receives from each slave node the partial error function and gradient, and accumulates them to find the complete error function and gradient values.

Note that the error function values are sent to the master only if the employed training algorithm uses them; otherwise only the partial gradient values are communicated. Finally, the weights are updated (using any batch training algorithm), and the new weights are sent to the slave nodes for the next epoch. When the termination condition is fulfilled, the master node sends termination signals to the slaves and shuts the PVM down. Below, a high level description of the master algorithm is presented.

The process of receiving the partial error function and gradient values can be realized in a synchronous or an asynchronous mode. When the synchronous mode is selected the master is forced to communicate with the slaves in a specific order. On the other hand, in asynchronous mode the communications are performed in a first–come, first–serve basis. Obviously, the asynchronous mode is the preferred one, since the master does not have to wait for a slower slave, but instead can continue gathering information from the other slaves.

```
Procedure master
InitializeAllSlaves;
InitializeMLP;
PartitionTrainingSet;
For slave := 1 to numOfSlaves do
   SendMLP;
   SendPatterns;
Repeat
   For slave := 1 to numOfSlaves do
      RcvPartialErrorFunction;
      RcvPartialGradient;
   AccumulateErrorFunction;
   AccumulateGradient;
   AdaptWeights;
   For slave := 1 to numOfSlaves do
      SendWeights;
Unitl TerminationCondition;
ShutDownSlaves;
ShutDownPVM;
```

3.2 The Algorithm of the Slave Nodes

Each slave node initially receives the network architecture and its part of the training set. Then, it calculates the partial error function and gradient values, by means of a forward and a backward pass, and sends them to the master node. Finally, the master node sends the updated weights for the next epoch. Below, we provide a high level description of the slave algorithm.

```
procedure slave
RcvMLP;
RcvPatterns;
Repeat
   CalculatePartialErrorFunction;
   CalculatePartialGradient;
   SendPartialErrorFunction;
   SendPartialGradient;
   RcvWeights;
Until ShutDown;
```

4 Experimental Results

To demonstrate the efficiency of the proposed methodology, we have tested it on a difficult, real–life classification task; the detection of malignant regions in colonoscopic video sequences. Textures from normal and abnormal tissue samples have been randomly chosen from four video frames of the same sequence, and used for training in parallel the network to discriminate between malignant and normal regions (see [6] for further technical details).

Using the proposed parallel methodology, the following batch training algorithms have been tested: the BP, the momentum BP (MBP) [5], the adaptive BP (ABP) [13], the Non-Monotone Back-Propagation with Variable Stepsize (NMBPVS) [8, 11] and the Non-Monotone Barzilai–Borwein back-Propagation NMBBP [11] algorithms. The algorithms have been tested using the same initial weights, initialized by the Nguyen–Widrow method [9], and received the same sequence of input patterns. The weights of the network are updated only after the entire set of patterns to be learned has been in parallel presented and processed.

To generate the training set the co–occurrence matrices have been used. More specifically, the endoscopic image was separated into windows of size 16 pixels by 16 pixels. Then the co–occurrence matrices algorithm was used to gather information regarding each pixel in an image window [6]. Co–occurrence matrices, [4], represent the spatial distribution and the dependence of the gray levels within a local area.

So in our experiments, the feature vectors contain sixteen elements each and therefore the first layer of the MLPs will consist of sixteen neurons. An MLP having 16 inputs, 30 hidden and 2 output nodes has been trained to discriminate between normal and abnormal image regions using 1200 randomly selected patterns from four video frames. The training procedure stopped when the MLP exhibited 3% misclassifications on the training set.

Table 1 summarizes the performance of the algorithms for simulations that reached solution. The reported parameters are: *min* the minimum number of epochs, *mean* the mean value of epochs, *max* the maximum number of epochs, *s.d.* the standard deviation, and *succ.* the simulations succeeded out of 100 trials.

Algorithm	min	mean	max	s.d.	succ.
BP	7697	8505.5	9314	1143.39	20%
BPM	5685	8500.0	9315	952.58	32%
ABP	453	734.1	1055	249.37	99%
NMBBP	261	374.7	515	129.08	100%
NMBPVS	263	656.3	955	227.24	100%

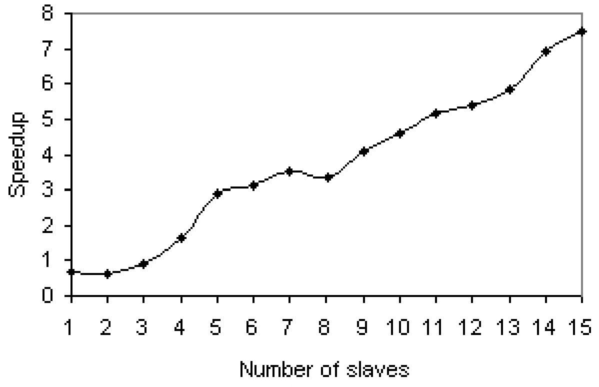
Table 1: Simulation Results.

To test the generalization performance of the trained MLPs approximately 16,000 test patterns have been created. This test set constitutes the whole image region in each of the four frames and contains normal and abnormal samples. In Table 2, the generalization capability of the algorithms, on the test set, is exhibited. Better generalization results for this difficult problem can be achieved using specially designed training algorithms (see for example [7]).

Algorithm	Generalization
BP	78.09%
BPM	78.09%
ABP	79.10%
NMBBP	83.91%
NMBPVS	85.12%

Table 2: Generalization Results.

Finally, we have tried to determine the speedup of the parallel procedure. Several factors can influence the speedup, such as the network load and the CPU load due to system or other users' tasks. However, speedup results indicate that when using more than three slave nodes the combined processing power of the PVM overbalance the overhead due to its initialization and process communication. Thus, in this case a speedup is always possible. In the following figure the speedup versus the number of processors is plotted.



The results indicate that the speedup is considerable and worth the minimal effort of parallelizing the learning algorithm, although it is not analogous to the number of slaves used. Obviously, this was expected due to the overhead introduced by the network and the PVM itself.

5 Conclusions

The combined computational resources of several general–purpose workstations, interconnected with a local area network have been exploited to implement parallel neural network learning algorithms. The proposed methodology has large granularity and has exhibited low synchronization, since frequent process synchronization was unnecessary. Furthermore, PVM's performance was stable and predictable. Notice that once built, the PVM can be used for any CPU intensive computational task (for example see [10]).

Simulation results have shown that speedups are always possible and justify the extra effort of parallelizing the learning algorithm, especially when large network architectures and training vectors are used.

Quantity	*Item*	*Unit price*	*Total*
15	Pentium III 667 Mhz processor and appropriate motherboard	$500	$7,500
15	128 MB of SDRAM	$40	$600
15	10/100 Mbps Ethernet network interface card	$10	$150
15	Midtower case with 230 Watt power supply and fan	$45	$675
1	10/100 Mbps Ethernet 24–port Switch	$330	$330
16	Ethernet cables	$5	$80
1	Master computer with Pentium III 667 Mhz, 128 MB of RAM, 10/100 Mbps Ethernet network card, 10 GB hard disk, video display, mouse, monitor, keyboard	$650	$650
16	Linux Operating System	$0	$0
		TOTAL	$9,985

Table 3: Cost of materials for a 15–node system

Appendix 1

In this Appendix we exhibit the cost of materials for a 15–node system. As can be seen from Table 3, the total price for the entire system is less than $10,000. It must be noted that only the master node needs hard disk, video display, monitor and keyboard. This is possible because of the use of *Beowulf–style* nodes. Additional information about the Beowulf Project and implementation details for parallel computer systems, can be found on the WWW at: `http://www.beowulf.org`.

The operating system used was Linux, which is the most common operating system for individual nodes of Beowulf–style parallel computer systems. Another reason for one to choose Linux is that it is open–source (its source code is provided) and free; no licence is required.

References

[1] The Beowulf Project, http://www.beowulf.org, last accessed 15/05/2001.

[2] L. Coetzee and E.C. Botha, An analysis of coarse–grain parallel training of a neural net, *Network: Computation in Neural Systems*, **6**, 73–91, (1995).

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, (1994).

[4] RM. Haralick, Statistical and structural approaches to texture, *IEEE Proc.*, **67**, 786–804, (1979).

[5] R.A. Jacobs, Increased rates of convergence through learning rate adaptation, *Neural Networks*, **1**, 295–307, (1988).

[6] S. Karkanis, G.D. Magoulas, and N. Theofanous, Image recognition and neuronal networks: Intelligent systems for the improvement of imaging information, *Minimal Invasive Therapy & Allied Technologies*, **9**, 225–230, (2000).

[7] G.D. Magoulas, V.P. Plagianakos, and M.N. Vrahatis, Hybrid Methods Using Evolutionary Algorithms for On–line Training, In *Proc. of the IEEE International Joint Conference on Neural Networks (IJCNN'2001)*, Washington D.C., (2001).

[8] G.D. Magoulas, M.N. Vrahatis, and G.S. Androulakis, Effective back–propagation with variable stepsize, *Neural Networks*, **10**, 69–82, (1997).

[9] D. Nguyen and B. Widrow, Improving the learning speed of 2-layer neural network by choosing initial values of the adaptive weights, In *Proc. of the IEEE First International Joint Conference on Neural Networks*, **3**, 21–26, (1990).

[10] V.P. Plagianakos, N.K. Nousis, and M.N. Vrahatis, Locating and Computing in Parallel all the Simple Roots of Special Functions Using PVM, *Journal of Comp. and Applied Mathematics*, in press, (1999).

[11] V.P. Plagianakos, M.N. Vrahatis, and G.D. Magoulas, Nonmonotone Methods for Backpropagation Training with Adaptive Learning Rate, In *Proc. of the*

IEEE International Joint Conference on Neural Networks (IJCNN'99), Washington D.C., (1999).

[12] T.L. Sterling, J. Salmon, D.J. Becker, and D.F. Savarese, *How to build a Beowulf: A Guide to Implementation and Application of PC Clusters*, MIT Press, Cambridge, (1999).

[13] T.P. Vogl, J.K. Mangis, J.K. Rigler, W.T. Zink and D.L. Alkon, Accelerating the convergence of the back-propagation method, *Biological Cybernetics*,**59**, 257–263, (1988).