A PARALLEL IMPLEMENTATION OF BACKPROPAGATION NEURAL NETWORK ON MASPAR MP-1[1]

Faramarz Valafar
Center of Complex Carbohydrate Research
University of Georgia
220 Riverbend Road, Athens, GA 30602
faramarz@mond1.ccrc.uga.edu
Fax: (706) 542-4412

Okan K. Ersoy
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47907
ersoy@amalthea.ecn.purdue.edu
Tel.: (317) 494-6162

Editor: Sara M. Valafar

Keywords: Parallel, SIMD, Neural Networks, Backpropagation, Delta Rule

ABSTRACT

In this paper, we explore the parallel implementation of the backpropagation algorithm with and without hidden layers on MasPar MP-1. This implementation is based on a SIMD architecture, and uses a backpropagation model. Our implementation uses *weight batching* versus *on-line* updating of the weights which is used by most serial and parallel implementations of backpropagation. This method results in a smoother convergence to a solution which is comparable to that of the popular method.

Versus various systolic array implementations of the backpropagation algorithm which are data driven, and exploit pipelined parallelism, we have developed a true SIMD algorithm which is control driven and exploits two types of parallelism inherent in backpropagation feedforward, layered neural networks, namely *architectural parallelism* and *data parallelism*.

Most importantly, the processing time is reduced both theoretically and experimentally by the order of 3000 for a network with 7-100-10 input-hidden-output neurons and 1188 training. With this algorithm we have achieved speeds of up to 70 Million Connections per Second (MCS) as throughput of a network stage and over 180 Million Connection UPdates per Second (MCUPS) for training the above network. This is the fastest performance of the standard backpropagation algorithm reported to date.

[1] The Purdue University MasPar MP-1 research is supported in part by NSF Parallel Infrastructure Grant #CDA-9015696

1. INTRODUCTION

Parallel implementation of neural networks has recently become a focus of research. Many different models of neural networks have been studied and their potential parallelism investigated. One such model which has enjoyed popularity is the backpropagations model [1], [16]. This model has been used to train multi-layered feed-forward neural networks.

Because of the inherent data-driven characteristics of backpropagation neural networks, their systolic array[2-3] implementation has been studied extensively. These studies can be categorized in the following three classes: 1. Mapping of the systolic array algorithm onto a parallel computer such as Wrap, MasPar MP-1, or Transputer arrays [4-8]. 2. Designing programmable systolic arrays for general neural network models [9-12]. 3. Designing a VLSI systolic array dedicated to only a few specific models [13-15]. All of these approaches map the neural network learning algorithm to a systolic array algorithm.

In this paper, we discuss a purely SIMD [2-3] mapping of the multi-layered feed-forward backpropagation algorithm. This algorithm creates and trains a separate network for each pattern in parallel. The interconnection weights are updated by an amount based on the total weight change summed over all networks. This technique is called weight batching [1] and exploits data parallelism as well as architectural parallelism of the backpropagation neural network algorithm versus the pipelined parallelism exposed by the systolic array implementation of the on-line method [1].

We discuss the implementation of this SIMD algorithm on MasPar MP-1 [17-19]. The MP-1 is configurable with up to 16K Processing Elements (PEs). We assume a fully configured MP-1.

The paper consists of four sections. Section 2 discusses the serial backpropagation algorithm, the parallel version of the backpropagation algorithm referred to as the SIMD-BP, and its implementation on MasPar MP-1. Section 3 discusses an investigation of the speed-up factor of the SIMD-BP algorithm as compared to the serial backpropagation implementation, the actual speed-up achieved by MP-1, and other related issues. Section 4 is the conclusion.

2. THE SERIAL BP AND THE SIMD-BP ALGORITHMS

The parallel version of the backpropagation algorithm (referred to as SIMD-BP) is designed for MasPar MP-1 with 16K PE's. Our design included backpropagation networks with one and no hidden layer. Without any hidden layer, the algorithm is the same as the *delta rule* [1] with output layer nonlinearities.

The first step is to modify the backpropagation algorithm so that it can be implemented in a SIMD fashion. In standard backpropagation, an input pattern is presented to the network. Based on that pattern, the network computes an output pattern. The output pattern is compared to a desired pattern and an error vector is computed. The error is backpropagated through the network; based on the amount of error passing through each connections, the weights are changed. After that, the next pattern is presented to the network and this procedure is repeated for the new pattern. In the SIMD version of this algorithm, the weights are not changed after each pattern. The weight changes are stored; after the

completion of a sweep[2], they are added together and only then the weights are updated, based on the total weight change computed.

Let us assume a network with *N* output neurons for a problem with *P* training patterns. The backpropagation algorithm attempts to find a minimum for the total squared error defined for one training sweep[1]. The weights are updated as follows:

$$\Delta w_{ij} = \rho \sum_{p=1}^{P} x_j^p o_i^p \left(1 - o_i^p\right)\left(d_i^p - o_i^p\right) \qquad (1)$$

$$\Delta v_{jk} = \rho \sum_{p=1}^{P}\left[i_k^p x_j^p \left(1 - x_i^p\right) \sum_{n=1}^{N} w_{nj} o_n^p \left(1 - o_n^p\right)\left(d_n^p - o_n^p\right)\right] \qquad (2)$$

Where d_n^p is the desired output value of the nth output neuron for the pth training pattern, o_n^p is the actual output of the nth neuron for the pth training pattern, Δw_{ij} and Δv_{jk} are the second and first stage weights respectively, ρ is a sufficiently small step size, and x_j^p and i_k^p are the jth and kth inputs to the second and first stage respectively.

In other words, the network has to calculate the weight changes due to all the training patterns, add them up and update the weights based on the total weight change accumulated over the entire sweep. In practice, however, in the serial implementation, the weight update is performed after each training pattern (on-line method). In other words, using (*1*) and (*2*), the weight changes are computed as

$$\Delta w_{ij} = \rho x_j^p o_i^p \left(1 - o_i^p\right)\left(d_i^p - o_i^p\right) \qquad (3)$$

$$\Delta v_{jk} = \rho\left[i_k^p x_j^p \left(1 - x_i^p\right) \sum_{n=1}^{N} w_{nj} o_n^p \left(1 - o_n^p\right)\left(d_n^p - o_n^p\right)\right] \qquad (4)$$

It can be shown that if the step size ρ is sufficiently small, the weight update can be performed after each pattern and reach a minimum of the error function E after a series of very small steps. While this approach is shown to work, its speed is very slow.

The SIMD-BP uses the exact method, mainly because it allows data parallelism. Each network computes a weight change vector for all the weights in the network, based on the training pattern it is given. After the sweep is complete, these weight change vectors are added together using a very fast MP-1 library routine called *reduceAdd* [17, 19]. Then, the weight vectors on all the networks are updated based on the total weight change vector. This vector is sent to all the PEs of MP-1 using the *XNET* [17, 19] structure.

The use of the exact algorithm results in data parallelism. Most of the speed-up achieved is due to this type of parallelism. Thus, the two types of parallelism utilized by the SIMD-BP are as follows:

1. **Architectural Parallelism:** This parallelism is simply due to the parallel nature of the architecture of the multistage network. The computations performed in the neurons of the same stage can be performed all at the same time. Since there are no connections between the

[2] Sweep is one round (sweep) of training over all patterns in the training set.

neurons of the same stage, no communication overhead is necessary.

2. **Data Parallelism:** As discussed above, most of the speed-up is due to data parallelism. Since the weight changes do not occur until after the sweep is over, there is no more data dependency between the operations performed for different patterns in the sweep. Consequently, these computations can all be done in simultaneously. Therefore, we can simulate more than one network at a time and train each one with a different input pattern, in parallel. These networks all have the same initial random weights and, ideally, only one input pattern to learn. Each network calculates updates for its weights based on the input pattern and the desired output pattern it is assigned to. This is done for all the networks at the same time. After this step, the weight changes are accumulated from all the networks and added in the ACU (Array Control Unit) [17] of MP-1. The weights of all the networks are updated simultaneously, based on the total weight changes.

To better describe the SIMD-BP training algorithm, we discuss the algorithm with the example of the 10-class remote sensing Colorado problem. This problem was described in [21]. It involves classifying each input pattern into one of ten possible classes. The data set consists of 1188 patterns of length seven for training and 831 patterns for testing.

Each network is simulated by 100 PEs, which is the size of the hidden layer of the backpropagation network. These 100 PEs, first emulate the 100 hidden neurons of the network. Once the calculations for the first stage are performed, the output values of the 100 hidden neurons are communicated to the first 10 of the 100 PEs. Then, the remaining 90 are disabled and only the first 10 PEs are active to emulate the output layer..

It is important to keep in mind that the degree of parallelism achieved depends on the number of processors assigned to each network, and the number of patterns in the training set, as well as parallel resources of the machine. For example, the 10-class Colorado problem has 1188 patterns in its training set and the number of PEs required for each network is 100. Therefore, the maximum number of networks running simultaneously is $\leq \frac{16384}{100} = 163$. For the simplicity of communication patterns, we chose to have only 156 networks running simultaneously.

Out of 156 networks, 94 were given 8 patterns and the remaining 62 were given 7 patterns (7x62+8x94=1188), which gives a degree of virtualization of 8 which is explained further below. Hence, at any given time, we are computing the weight changes for 156 different patterns.

In any parallel machine, the degree of parallelism is limited to the physical parallel resources of the machine. For example, in MP-1 with 16K PEs, the maximum degree of parallelism achievable is 16384, since a maximum of 16384 operations can be run in parallel at any given time. The real degree of parallelism for a given algorithm is normally much less than the maximum degree possible. For example, in the Colorado problem, every network required 100 PEs, thus allowing 156 parallel networks. In order to have one network per training patterns, we ideally would have required 100x1188=118800 PEs. Since this many PEs were not available, we implemented a concept referred to as *virualization*. The idea is similar to that of virtual memory, where one assumes that there is a much larger memory space than what the machine's physical resources offer. We assumed that 118800 PEs were arranged in a three dimensional PE grid array. The three-dimensional array is made of 8 layers (*slices*) of 128x128 PEs. Since there is actually one physical layer of PEs available, the PE

array of MP-1 has to be programmed to emulate the layers of the 3-D grid serially. Thus, we end up running 156 networks at a time and, at any given time, the PE array is emulating a different layer of the virtualized PE grid.

Another costly part of initiating the 156 networks is the generating floating point random numbers for initial connection weights and distributing them among the PEs correctly. This procedure is so costly that storing some random values and loading them from a file should be considered. To generate the random numbers, we used a random vector generator routine from the MasPar mathematics library called *fp_veyran* [17] which generates a Y-oriented random vector and stores its elements in the first column of the MP-1 PE array. To distribute the weights among all networks, we, again, used the *xnet* constructs. Table 1 shows the average time required for this task.

Table 1. Actual Time Indexes for Various Parts of the SIMD-BP Algorithm.

			First Stage	Second Stage	Overall Network
Throughput	Best Time	MCS	73.12	19.26	28.55
		Sec./Sweep	1.3001e-2	6.2314e-2	7.5315e-2
	Worst Time	MCS	73.07	19.25	28.54
		Sec./Sweep	1.30064e-2	6.23242e-2	7.53306e-2
Weight Update	Best Time	MCUPS	186.94	39.47	60.60
		Sec./Sweep	5.084e-3	3.0401e-2	3.5485e-2
	Worst Time	MCUPS	186.72	39.33	60.40
		Sec./Sweep	5.09008e-3	3.05117e-2	3.5611e-2
Loading and Distributing Training Data			0.236411 Seconds for 1188 Patterns 5025.15 Patterns/Second		
Loading and Distributing Desired Data			4.10522e-2 Seconds for 1188 Patterns 28938.77 Patterns/Second		
Generating and Distributing Random Weights			0.449686 Seconds for 1810 Connections 4025.03 Connections/Second		

The SIMD-BP program is designed to arrange the PE array to achieve the minimum degree of virtualization, thereby achieving the maximum degree of parallelism. It is written in a way that it detects and adjusts to the size of any given problem automatically. For this purpose, the program considers three parameters: 1) The size of the largest layer of the network, 2) The number of training patterns, and 3) The actual size of the PE array. Both in training and in testing, the SIMD-BP takes the degree of virtualization (*slice*) and a parameter called *offset* into account. The *offset* is the number of PEs in the last slice which still have data and should be kept active for the calculations of each slice. The program then performs the operations of each slice serially. It first deactivates the PEs not required for that slice and then has the ACU (see Appendix) decode the instructions and send them to the PEs, which in turn perform the operation if their enable flag is high. The SIMD-BP program, thereby, is written in a way that it detects and adjusts to the size of any given problem and PE Array automatically.

The way the networks are organized is such that the first PE in all the networks can easily be enabled. The input patterns are loaded into the first PEs of the networks using the parallel read command [17]. After the loading of input data, the first PEs proceed to communicate the data to the rest of the PEs in their networks.

3. TIME COMPLEXITY ANALYSIS AND ALGORITHM PERFORMANCE

In this section, we analyze the time complexity of the serial backpropagation (BP) and the SIMD-BP algorithms. Since most of the required computation time for any network is used to train the network, we only concentrate on the time complexity of respective training procedures.

Since the time taken to perform floating point addition, multiplication, and exponentiation is a good indication of the time required by the training procedure, we estimate the number of such operations to be performed in each type of training procedure.

The Serial BP Algorithm:

Let us denote the number of input neurons of the network with n_i, the number of hidden neurons with n_h (assuming one hidden layer), the number of output neurons with n_o and the number of patterns in the training set with *P*. It can easily be shown that the time complexity of the serial backpropagation algorithm is in the order of:

$$T_{BP} = O\left(Pn_h^2\right) \tag{5}$$

The SIMD-BP Algorithm:

To express the time complexity of the SIMD-BP, in addition to the time required for floating point additions, multiplications, and exponentiations, we have to consider the communication overhead. Let us first consider the additions, the multiplications, and the exponentiations. Since in SIMD-BP all the neurons of each stage operate in parallel, we only need n_i additions, n_i multiplications, and 1 exponentiation for the first stage and n_h additions, n_h multiplications, and 1 exponentiation for the second stage. Let $\alpha, \mu,$ and γ be the time required for one floating point addition, multiplication, and exponentiation respectively. Since the communication overhead is on the order of the length of a side of the PE array which is 128, the communication overhead is on the order of $nyproc \times \varsigma$, where *nyproc* is the length of one side of the PE array, and ς is the time it takes to communicate a float value from one PE to its immediate neighbor. Hence, we get:

$$T_{SIMD-BP} = O\left[\left[\left(n_i + n_h\right)(\alpha + \mu) + 2\gamma + nyproc \times \varsigma\right] slice\right] \tag{6}$$

where $slice = \left\lceil \frac{P \times n_h}{N} \right\rceil$, is the degree of virtualization and *N* is the number of PEs in the MP-1 PE array. Because both n_i and n_h are $O(nyproc)$, we can write

$$T_{SIMD-BP} = O\left[\frac{Pn_h\left[\left(n_i + n_h\right) + nyproc\right]}{N}\right] = O\left(\frac{Pn_h\, nyproc}{N}\right) = O\left(\frac{Pn_h}{\sqrt{N}}\right) \tag{7}$$

where $N=nyproc^2$. The order of estimated speed-up, using (*5*) and (*7*) is then measured by:

$$O\left(\frac{T_{BP}}{T_{SIMD-BP}}\right) = O\left(\frac{Pn_h^2}{\frac{Pn_h}{\sqrt{N}}}\right) = O\left(n_h\sqrt{N}\right) \qquad (8)$$

For the example of the 10-class Colorado Problem, the above ratio becomes $O\left(100 \times \sqrt{16384}\right) = O(12800)$. In our experiments with backpropagation on a SPARC5 work station, each sweep of training for the 10-class problem took an average of approximately 7 minutes and 30 seconds. On MP-1, every 100 sweeps took an average of approximately 14 seconds. This results in a speed-up factor of 3214 in this particular case.

It is important to mention here that this speed-up factor embodies both parallel speed-up and hardware differences in the floating point units of the two systems. The floating point unit in the SPARC is a full blown coprocessor, where the floating point units of MP-1 have 4-bit ALUs and most of their operations are performed by table look-ups. In addition, in MP-1 the floating point units are shared among the PEs of the same PE cluster. (This is not the case in MP-2). Figure 1 shows the run times for different size hidden layered networks, emulated by SIMD-BP. The relatively big jump in the training time between 80 and 90 hidden neurons is due to the addition of another slice to the virtual PE array and thereby increasing the degree of virtualization by 1.

4. CONCLUSIONS

Implementing neural network algorithms in massively parallel machines is very promising to reduce the training time from hours to minutes. This kind of speed-up is impossible to achieve even with a fast neural network algorithm implemented on the fastest serial machine.

The backpropagation algorithm offers architectural parallelism and data parallelism in the way it is parallelized in this article. While the degree of architectural parallelism is limited to the size of the largest layer of the network, the degree of data parallelism is only limited by the number of the PEs and the number of training patterns, which are both often far larger than the number of the neurons in the largest layer of the network.

Massively parallel implementations of neural networks allow larger problems to be investigated in a short period of time. Since the properties of neural networks often arise by the collective behavior of all the neurons, such implementations also have the potential of helping in the understanding of artificial and biological mechanisms of intelligence.

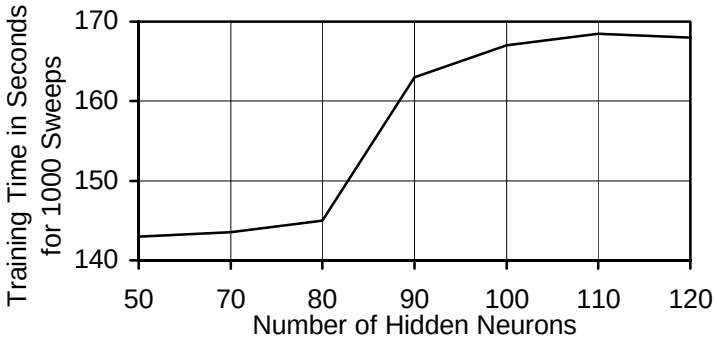

Figure 1. SIMD-BP Run Times for Networks with 7-10 Input-Output Neurons for the Colorado Data Set with 1188 Training Patterns

REFERENCES:

[1] D.E. Rumelhart, J.L. McClelland, *Parallel Distributed Processing*, The MIT Press, Cambridge Massachusetts, 1986.
[2] K. Hwang, F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Computer Science Series, New York, 1984.
[3] G.S. Almasi, A. Gottlieb, *Highly Parallel Computer*, The Benjamin/Cummings Publishing Company, Inc. 1990.
[4] D.A. Pomerleau, G.L. Gusciora, D.S. Touretzky, and H.T. Kung, "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second", *Proc. IEEE International Conf. on Neural Networks*, Vol. II, San Diego, CA, pp. 143-150, July 1988.
[5] S. Borkar et al., "iWarp: An Integrated Solution to High Speed Parallel Computing", *Proc. Supercomputing '88, IEEE Computer Society*, Orlando, FL, pp. 330-339, 1988.
[6] J.R. Millan, P. Bofill, "Learning by Back-propagation: A Systolic Algorithm and Its Transputer Implementation", *Neural Networks 3*, pp. 119-137, July 1989.
[7] R. Mann, S. Haykin, "A Parallel Implementation of Kohonen Feature Maps on the Warp Systolic Computer", Proc. *International Joint Conference on Neural Networks*, Vol. II, Washington, DC, pp.84-87, Jan. 1990.
[8] G. Chinn, K.A. Grajski, C. Chen, C. Kuszmaul, and S. Tomboulian, "Systolic Array Implementations of Neural Networks on the MasPar MP-1 Massively Parallel Processor", *Proc. International Joint Conference on Neural Networks*, Vol. II, San Diego, CA, pp. 169-173, June 1990.
[9] S.Y. Kung, J.N. Hwang, "Parallel Architectures for Artificial Neural Nets", *Proc. International Joint Conference on Neural Networks*, Vol. II, San Diego, CA, pp.165-172, July 1988.
[10] H. Kato, H. Yoshizawa, H. Iciki, and K. Asakawa, "A Parallel Neurocomputer Architecture Towards Billion Connection Updates Per Seconds", *Proc. International Joint Conference on Neural Networks*, Vol. II, Washington, DC, pp.51-54, Jan. 1990.
[11] U. Ramacher, W. Raab, "Fine-grain System Architectures for Systolic Emulation of Neural Algorithms", *Proc. International Joint Conference on Application Specific Array Processors, IEEE Computer Society Press*, pp. 554-566, Sept. 1990.

[12] A. Hiraiwa, M. Fujita, S. Kurosu, S. Arisawa, and M. Inoue, "Implementation of ANN on RISC Processor Array", *Proc. International Joint Conference on Application Specific Array Processors, IEEE Computer Society Press*, pp. 677-688, Sept. 1990.
[13] F. Blayo, P. Hurat, "VLSI Systolic Array Dedicated to Hopfield Neural Network", *Artificial Intelligence*, Kluwer Dordrecht, pp. 255-264, 1989.
[14] F. Blayo, C. Lehmann, "A Systolic Implementation of the Self Organization Algorithm", *Proc. International Neural Network Conf*., Vol. II, pp. 600, Paris, July 1990.
[15] H.K. Kwan, P.C. Tsang, "Systolic Implementation of Multi-Layer Feed-Forward Neural Network with Backpropagation Learning Scheme", *Proc. International Joint Conference on Neural Networks*, Vol. II, Washington, DC, Jan. 1990.
[16] P.J. Werbos, "Backpropagation: Past and Future", *Proceedings of ICNN 88*, San Diego, CA, pp. 343-353, June 1988.
[17] MasPar Mp-1 Reference Manuals, MasPar Computer Corporation , Sunnyvale, CA
[18] Kenneth E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transaction on Computers*, Vol. C-29, pp. 836-840, Sept. 1980.
[19] Peter Christy, "Software to Support Massively Parallel Computing on the MasPar MP-1", *Proceedings of the IEEE Compcon Spring 1990*, Feb. 1990.
[20] P.J.B. Hancock, "Data Representation in Neural Nets: An Empirical Study", *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann Publishers Inc., pp. 11-20, 1988.
[21] O.K. Ersoy, D. Hong, "Parallel, Self-Organizing, Hierarchical Neural Networks", *IEEE Tran. on Neural Network*, Vol. I, No. 2, pp. 167-178, June 1990.
[22] D.G. Luenberger, *Linear and Nonlinear Programming*, Second Edition, Addison-Wesley Publishing Company, Massachusetts, 1984.