Massive Parallelism in Neural Network Simulation

Andreas Zell, Niels Mache, Markus Hüttel, Michael Vogt
University of Stuttgart,
Institute for Parallel and Distributed High Performance Systems (IPVR),
Breitwieserstra. 20-22, D-7000 Stuttgart 80,
Fed. Rep. Germany
E-mail: zell@informatik.uni-stuttgart.de

Abstract: We here present and compare different massively parallel implementations of multilayer feedforward neural networks on a MasPar MP-1216, a parallel SIMD computer with 16,384 processors. For multilayer feedforward networks we have obtained sustained rates of up to 348 M CPS and 129 M CUPS with backpropagation, a high mark for this architecture and for general purpose SIMD computers. This paper focuses on the problems of mapping neural networks to parallel hardware, on implementation problems in obtaining high propagation rates on a SIMD machine and on problems with the resulting learning algorithms.

Keywords: connectionism, neural networks, neural network simulators, massive parallelism, simulation

1 Introduction and Motivation

In the last five to eight years, the focus of research in artificial intelligence (AI) has shifted from a purely symbolic view to a view that encompasses subsymbolic or connectionist systems as another powerful computing method for solving problems in artificial intelligence [Rumelhart, McClelland 86]. Connectionist systems or artificial neural networks (ANNs) consist of a large number of simple units (cells, artificial neurons) working in parallel and exchanging information via a network of directed, weighted links (connections). This information usually only comprises the activation of the neurons, a single numerical value for each cell, which is fed to the successor cells after being weighted by their connecting links. This is motivated by a rough analogy with the synaptic coupling of nerve cells. The information that the network has learned after training is usually distributed over the network in the form of the matrix of all link weights.

Our research group wants to understand the advantages and the trade-offs of the various artificial neural network paradigms and learning algorithms, their efficiency and generalization capabilities and their suitability for massively parallel implementation. We have developed a neural network simulator, SNNS, which has proven well suited for research on learning algorithms, on issues of visualizing network topology, training and performance and on parallel implementation of neural networks. It is also used in a number of other university research groups and with growing acceptance in industry as a neural network evaluation and prototyping tool. In this paper we are describing the experiences we gained in developing a massively parallel simulator kernel for SNNS running on our 16 K processor MasPar MP-1216.

2 Stuttgart Neural Network Simulator

SNNS (Stuttgart Neural Network Simulator) is an efficient and portable neural network simulation environment for Unix workstations developed at the Institute for Parallel and Distributed High Performance Systems, University of Stuttgart, Germany. It is a software tool to generate, train, test and visualize artificial neural networks. The whole network simulator has been developed in C on Unix workstations. The graphical user interface was implemented under X-Windows X11R4 (Athena widget set), for maximal portability.

2.1 Structure of SNNS

SNNS now consists of a sequential and a parallel simulator kernel and a graphical user interface. The simulator kernel operates on the internal representation of the neural networks and performs all operations of the learning and recall phase. It is closely coupled with the graphical user interface via an interface of function calls. The simulator kernel is written in C for efficiency and portability and has already been ported to a number of architectures.
Fig. 1 Structure of the SNNS network simulator consisting of sequential simulator kernel, parallel simulator kernel and graphical user interface

The graphical user interface, based on X-Windows, is a powerful tool to construct the network topology and visualize and modify small to medium sized nets interactively. It can also be used to generate and save test patterns for small networks. To economize on screen space the display elements are kept in separate windows and thus can be arbitrarily arranged or hidden if desired. There are various ways to display or modify nodes and links or selected sets of them. An integrated help facility aids the novice with the interface. Networks can be modified through the user interface during simulation. Units can be introduced, removed, or have their activation values changed. Connections among the units can be inserted, deleted, redirected, or have their strengths modified. Contrary to most other simulators most modifications can be done in a very simple point and click manner directly from the visual representation of the network topology.

2.2 Graphical User Interface

The graphical user interface, based on X-Windows, is a powerful tool to construct the network topology and visualize and modify small to medium sized nets interactively. All display elements are kept in separate windows and thus can be arbitrarily arranged or hidden if desired. There are various ways to display or modify nodes and links or selected sets of them. An integrated help facility aids the novice with the interface. Networks can be modified through the user interface during simulation. Units can be introduced, removed, or have their activation values changed. Connections among the units can be inserted, deleted, redirected, or have their strengths modified. Contrary to most other simulators most modifications can be done in a very simple point and click manner directly from the visual representation of the network topology. Fig. 2 gives an overview of the graphical user interface of SNNS.

Contrary to many other simulators most modifications can be done in a very simple point and click manner directly from the visual representation of the network topology. The user can control the visual representation of units (activation, output, number, name) and the display of links (directed, undirected, weight). Connections and units can be displayed selectively, i.e. the user may choose to display only those units whose activations or outputs exceed a given display threshold or only those links whose weights are in a certain range. This allows watching the growth of units and the change of link weights during learning.
The graphical interface is both used to display a neural network and to generate and manipulate it. Therefore, the user has a powerful set of operations (insertion, deletion, copying, moving) at his use. These operations may be applied to individual units or to selections of units and may affect links as well, like 'copy all selected units with their input links' or 'delete all links into the selected units'. These operations allow a quick and convenient generation of networks.

2.3 Connectionist Models supported by SNNS

From its design SNNS supports a wide variety of neural network models. Any network that can be specified as a directed graph with weighted links may be realized. The concept of sites which has been adapted from RCS [Goddard et al. 89] even allows multiple links between two single units. Most users of SNNS use simple multilayer feedforward networks with one or two hidden layers with standard sigmoid activation functions (logistic, sine or tanh). However, recurrent networks have also been implemented. The following learning algorithms have been implemented in SNNS: "vanilla" backpropagation [Rumelhart, McClelland 86], backpropagation with momentum and weight decay and flat spot elimination, batch backpropagation, quickprop [Fahlman 89], counterpropagation [Hecht-Nielsen 89], backpercolation [Jurik 89], cascade correlation [Fahlman 90], radial basis function networks (RBF) [Poggio, Girosi 89], ART1, ART2 and ARTMAP [Carpenter, Grossberg 88], Time-Delay Networks [Waibel 89] and self organizing feature maps. Not all of them are available in the public distribution, however.
2.4 Selected Applications of SNNS

SNNS is currently used in at least 300 installations worldwide, approx. one third of them each in Germany, other Europe and the U.S. Its main use is in university research but some commercial research projects use SNNS as a prototyping tool to find optimal learning procedures, network sizes and learning parameters for various neural network applications. Applications include rotation invariant pattern recognition, handwritten character recognition, stock price prediction, recognition and classification of exogenic and endogenic components of event correlated brain potentials, noise reduction in natural language communication in a telecom environment, prediction of secondary structure of proteins and texture analysis.

3 Massively parallel SNNS kernels on the MasPar MP-1

Two parallel implementations for the SNNS kernel and one prototype implementation have been developed on our 16 K processor MasPar MP-1216 for multilayer feedforward networks. The goal of the parallelization was to enable the simulation of large neural networks, mainly for the tasks of image processing, feature extraction and pattern and object recognition. The parallel simulator is integrated with the sequential simulator as an alternative simulator kernel. From the X-Windows based graphical user interface it is possible to switch between both kernels at runtime, provided the user restricts himself to multilayer feedforward networks.

3.1 Architecture of the MP-1

The MasPar MP-1216 is a SIMD machine with up to 16,384 four-Bit processors. 32 processors are integrated on a single chip, 32 chips fit on a processor board. Our full scale model delivers a quoted peak performance of 30,000 MIPS (32 bit addition) and 1,500 resp. 600 MFLOPS (32 bit resp. 64 bit). There exist two separate communication architectures on the MasPar: one is a 3-stage global router which allows up to 1024 simultaneous connections between any two processors, the other is a torroidal two-dimensional 8 neighbour grid (X-net). Communication bandwidth is up to 1.5 GB/s peak global router and up to 24 GB/s peak X-net communication. From this data it can be seen that it is advisable to use the local grid as much as possible since the communication bandwidth is much larger than with the router. Also on our machine we experienced relatively high router hardware failures which forced our implementations to avoid it if possible.

The MasPar can be programmed with parallel versions of C (AMPL) and Fortran. MPPE (MasPar parallel programming environment), an integrated graphical tool set based on X-Windows, facilitates program development and debugging. Having investigated the trade-offs of different approaches to parallelization of neural networks, as given in [Singer 90], [Grajski et al. 90], [Chinn et al. 90] and [Zhang et al. 89] we decided on an implementation which combines unit parallelism with training vector parallelism. All implementations of our parallel simulator kernel were done in MPL, a parallel extension of C. Two of them have recently been converted to AMPL, the ANSI C extension of MPL.

3.2 Implementation with Unit-Parallelism and Training Pattern Parallelism

The implementation of [Mache 92] uses the following technique (Fig. 3): All hidden and output units of a vertical slice are mapped to a single processing element (PE) of the MasPar. The computation of unit activation is done in parallel for all units of a layer. Thus, a number of processors is needed which equals the largest number of processing elements in a layer, i.e. the width of the network determines the number of processors needed. If the number of input units is greater than the number of units of the other layers (which is very often the case), an additional PE is needed to store the remaining components of the input pattern and to send them to its neighbor when they are needed. Each processor stores the weights of all of its input links. The processors are located in a logical ring communication structure which can easily be realized on the X-net grid (with possible copying at the fringes). During forward or backward propagation, the intermediate values for the net input or the accumulated error signal, resp., are shifted cyclically to the left. The weights are stored with a skew factor of 1 in each processor. This allows that all units of a layer perform the computation of the sum of all weighted predecessor units' outputs in a number of steps equal to the size of the preceding layer. The algorithm is very similar to a systolic matrix-vector multiplication algorithm.

Since the width of a feedforward network is usually much smaller than the number of available processors on our MasPar system, multiple copies of the network with different input patterns are kept in the machine and are updated in parallel. In this way weight changes have to be computed in each network individually without actually changing the weights. The sum of the weight changes is then computed and applied to all corresponding weights of the identical network copies. This results in a backpropagation algorithm that is a mixture between online and batch backpropagation with the batch size at least equal to the number of network copies in the machine or an integer multiple of it.
Fig. 3 Parallel MasPar SNNS kernel with a 6-3-4 feedforward network: all hidden and output neurons of a column and their input weights are mapped onto a single processor, all neurons of a layer are trained in parallel (unit parallelism). The number of processors needed is the number of neurons of the biggest layer except the input layer, plus one. Multiple network copies with different input patterns are trained in parallel (training pattern parallelism).

For an optimal 128-128-128 network which fits into the machine without an additional PE and which does not need copying at the end of a cycle this implementation we obtained 176 M CPS (connections per second) and 67 M CUPS (connection updates per second) for backpropagation training. The Nettalk network [Sejnowski, Rosenberg 86], a 203-120-26 network, can be trained with 41 M CUPS and operated with 98 M CPS. These times did not include the time for transfer of the input patterns from the frontend to the parallel machine.

One advantage of this approach is, that the numbers of processors used is not determined by the size of the input layer, which is usually much larger than any hidden or output layer. So a large number of networks can be trained in parallel. A disadvantage is the fact that the one additional PE has to store much more pattern elements than the others. In an SIMD machine with identical memory allocation on all PEs this memory becomes a limiting factor of how many patterns can be stored in parallel on the machine. Since pattern I/O was the limiting factor of our parallel implementation, a second implementation was performed.

### 3.3 Second Implementation with Unit-Parallelism and Training Pattern Parallelism

Our second implementation was done to alleviate the pattern I/O bandwidth problem of the first implementation. Its main objective was to store as many patterns as possible in the parallel PE memory, even if the number of PEs needed to store the network is larger. This implementation uses a number of PEs which is equal to the size of the biggest layer, including input layer. If the input layer is the biggest layer, all PEs store a similar number of pattern components, otherwise some PEs may store no components.

For an optimal 128-128-128 network we obtain sustained 348 M CPS in recall mode and 129 M CUPS for backpropagation training. The NETtalk network can be recalled with 47 M CPS and trained with 17.6 M CUPS. These times include the time for the transfer of the input patterns and the results. Since the I/O times dominated the learning and recall times in the previous implementation, the speed improvement of the latter version was even greater than the figures tell. This speed improvement resulted from a new, better compiler installed in the meantime, and from extensive code optimization. The fact that in this scheme less networks can be trained in parallel can be seen in NETtalk benchmarks, but it is far less important than the time saved by faster pattern loading.
Fig. 4 Second Parallel MasPar SNNS kernel with a 6-3-4 feedforward network: all neurons of a column and their input links are mapped onto a single processor. The number of processors needed is as large as the maximum number of neurons of any layer. Multiple network copies with different input patterns are trained in parallel.

3.4 Link-Parallel Implementation

The last implementation [Hüttel 92] is not a full SNNS kernel but was intended as a prototype implementation. It lacks the support of all SNNS kernel functions but can read SNNS network files. It is displayed in Fig. 3.

First the network is extended one bias unit for each layer and dummy units to make each layer of equal size n. All units of adjacent layers are connected. The weights to dummy units are initialized to zero and are prevented from being updated by masking them in the last step of weight updates. In our terminology weights from source i to j are denoted by \( w_{ij} \). If the weight matrices connecting adjacent layers are denoted \( W_1, \ldots, W_m \) then if \( r \) is odd, the outgoing weights \( w_{ij} \) of unit i are mapped to columns of the PE array, with the source unit of lowest index giving the leftmost column; if \( r \) is even, the outgoing weights \( w_{ij} \) of unit i are mapped to rows of the PE array, with the source unit of lowest index giving the bottom row. This parallel prototype implementation with link parallelism achieved 136 M CUPS for a fully connected 127-127-127 network and 160 M CUPS for a 127-127 network on our MasPar MP-1216.

4 Problems of the Parallel Simulator Kernels

All three parallel SNNS kernels on the MasPar yield impressive performance figures. However, these results have only been obtained after a lengthy period of optimization and after several complete rewrites of the parallel kernel. Our biggest hurdle was the slow communication of the training patterns from the unix workstation to the parallel backend, which in first tests took minutes versus milliseconds for the actual training. This could be improved a little with the use of a parallel disk array or the parallel I/O RAM that are available now as expensive options of the machine. A lot of effort was therefore spent to load training patterns in large blocks and to keep as many of them as possible in the distributed parallel PE memory.

Another problem concerns the batch backpropagation algorithm necessary to run the training pattern parallel implementations: For many applications with a large number of similar input patterns this learning algorithm is slower than online backpropagation. We tested our simulator with character recognition problems. In one case we used 10,000 scanned digits "0" to "9". In this test the slower convergence of batch backpropagation offset most of the performance gain of the parallel architecture. However, some applications need batch backpropagation for convergence and others report better generalization results. Also, other batch learning algorithms like quickprop [Fahlman 88] may be used with better results.
5 Conclusions

We here have investigated different mappings of neural networks to a massively parallel SIMD computer. These different implementations have shown that it is possible, albeit not at all easy to obtain impressive performance figures for neural network simulation on current SIMD computers. However, these high marks are only obtained for simple network architectures with a network size that fits well into the parallel machine. We have learned that propagation figures quoted for neural network algorithms are only meaningful if they take communication time from disk or workstation to the parallel machine into account. Overcoming the I/O bottlenecks took most of the time of the implementations and forced several fundamental changes in the algorithms. Our results can be extended to VLSI neural network hardware in the sense that the time to load training patterns into the parallel hardware must match the speed of propagation.

Another lesson learned was that the speed advantage gained by a parallel implementation can be lost for certain applications because of the slower batch backpropagation algorithm. These results have been obtained with precise floating point computations. It would have been more difficult with fixed point arithmetic or special VLSI hardware with limited precision.
The limited precision might itself offset the speed advantage of fast VLSI hardware.

Our last point is that these implementations are no natural mappings to parallel hardware, like e.g. each processor representing a neuron. Because of the limited communication bandwidth rather special mappings have to be found to obtain high performance on current hardware.

6 Literature


[Fahlman 88] Fahlman, S.E.: Faster Learning Variations on Backpropagation, in [Touretzky et al. 88]


[Hütte 92] M. Hütte: Parallele Implementierungen mehrstufiger feedforward-Netze auf einem SIMD-Parallelrechner, Studienarbeit Nr. 1124, Universität Stuttgart, Fakultät Informatik, Juli 92 (in German)

[Jurik 89] M. Jurik: Backpercolation, (unpublished) paper distributed by Jurik Research, PO 2379, Aptos, CA 95001 USA

[Mache 92] N. Mache: Entwicklung eines massiv parallelen Simulatorkerns für neuronale Netze auf der MasPar MP-1216, Diplomarbeit Nr. 845, Universität Stuttgart, Fakultät Informatik, Feb. 92 (in German)


241