3-1-1993

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

Faramarz Valafar
Purdue University School of Electrical Engineering

Okan K. Ersoy
Purdue University School of Electrical Engineering

A Parallel Implementation of Backpropagation Neural Network on Maspar MP-1

Faramarz Valafar
Okan K. Ersoy

TR-EE 93-14
March 1993

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907-1285

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

Faramarz Valafar
Okan K. Ersoy

School of Electrical Engineering
Purdue University
W. Lafayette, IN 47906

* The Purdue University MASPAR MP-1 research is supported in pan by NSF Parallel Infrastructure Grant #CDA-9015696.

ABSTRACT

One of the major issues in using artificial neural networks is reducing the training and the testing times. Parallel processing is the most efficient approach for this purpose.

In this paper, we explore the parallel implementation of the backpropagation algorithm with and without hidden layers [4][5] on MasPar MP-1. This implementation is based on the SIMD architecture, and uses a backpropagation model which is more exact theoretically than the serial backpropagation model. This results in a smoother convergence to the solution. Most importantly, the processing time is reduced both theoretically and experimentally by the order of 3000, due to architectural and data parallelism of the backpropagation algorithm. This allows large-scale simulation of neural networks in near real-time.

1. INTRODUCTION

Parallel processing is the most efficient approach to speed-up processing of algorithms whose calculations are, at least in part, independent from each other and can be performed simultaneously. A common form of parallelism is SIMD (Single Instruction Multiple Data) parallelism in which the same instruction is issued to all active processors. This instruction is then executed on all the processors simultaneously [1], [2], P I.

The MasPar MP-1 is a massively parallel computer which supports SIMD parallelism. The MP-1 has 16K Processing Elements (PEs) which can perform 16384 operations simultaneously. While all the processors work on the same operation, each PE uses its own data [1], [2].

Parallel processing of neural network algorithms is an important research issue since neural networks are large networks in practice, and they are used in applications which are often supposed to be real-time. One of the most commonly used neural network algorithms is backpropagation[4], [5].

Since the operations per neuron on each layer of neurons are independent of each other, the backpropagation algorithm can be implemented in the SIMD architecture. There is another type of parallelism called ***data parallelism*** in the backpropagation algorithm, which is discussed in Section 3.

The paper consists of six sections. Section 2 discusses the architecture of MP-1 in some detail as well as some software issues. Section 3 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 4 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 5 covers the SIMD delta rule algorithm, which corresponds to the SIMD-BP algorithm without hidden layers. Section 6 is conclusions.

2. INTRODUCTION TO MASPAR MP-1

Massively parallel computers commonly use more than 1024 processors to obtain computational speed unachievable by conventional computers. The MasPar MP-1 system is scalable from 1024 to 16384 processors and its peak performance scales linearly with the number of processors. A 16K processor system delivers 30,000 MIPS peak performance where a representative instruction is a 32-bit integer add. In terms of peak floating point performance, the 16K processor system delivers 1,500 MFLOPS in the single precision (32-bit) made and 650 MFLOPS in the double precision (64-bit)mode, in terms of the average of add and multiply times.

The MP-I has a Single Instruction Multiple Data (SIMD) architecture that simplifies the highly replicated processors by eliminating their instruction logic and instruction memory. The processors in a SIMD system are called Processing Elements (PE's).

The unique characteristics of the MP-1 architecture are the combination of a scalable architecture in terms of the number of Processing Elements (PE's), system memory, and system communication bandwidth; "RISC-like" instruction set design that leverages optimizing compiler technology; and an architectural design amenable to a VLSI implementation.

Figure 1 shows a block diagram of the MasPar system with five major subsystems. The following describes each of the major components:

The Array Control Unit (ACU): The ACU is a 14 MIPS scalar processor with a RISC-style instruction set. It fetches and decodes MP-1 instructions, computes addresses and scalar data values, issues control signals to the PE array, and monitors the status of the PE array. Most of the scalar ACU instructions execute in one 70 nsec clock. The ACU occupies one printed circuit board.

The ACU performs two primary functions: either PE array control or independent program execution. The ACU controls the PE array by broadcasting all PE instructions. Independent program execution is possible since it is a full control processor capable of independent program execution.

The ACU is a custom designed processor with the following major architectural characteristics:

— Separate instruction and data spaces

— 32-bit, two address, load/store, simple instruction set

— 4 Gigabyte, virtual, instruction address space, using 4K bytes per page.

The ACU has a microcoded implementation of its RISC-like instruction set due to the additional control requirements of the PE array. PE instructions typically require more than one clock cycle including floating point instructions which are well suited to a microcode implementation.

Processor Array: The MP-1 processor array (Figure 2) is configurable from 1 to 16 identical processor boards. Each processor board has 1024 PE's and associated memory arranged as 64 PE clusters (PEC's) of 16 PE's per cluster. The processors are interconnected via the X-Net neighborhood mesh and the global multistage crossbar router network. A processor board dissipates less than 50 watts; a full 16K PE array and ACU dissipate less than 1,000 watts.

A PE cluster (Figure 3) is composed of 16 PE's and 16 processor memories (PMEM). The PE's are logically arranged as a 4 by 4 array for the X-Net two-dimensional mesh interconnection. Each PE has a large internal register file shown in the figure as PREG. Load and store instructions move data between PRES and PMEM. The ACU broadcasts instructions and data to all PE clusters and the PE's all conmbute to an inclusive-OR reduction tree received by the ACU. The 16 PE's in a cluster share an access port to the multistage crossbar router.

The MP-1 processor chip is a full custom design that contains 32 identical PE's (2 PE clusters) implemented in two-level metal 1.6µ CMOS and packaged in a 164 pin plastic quad flat pack. The die is 11.6 mm by 9.5 mm, and has 450,000 transistors. A conservative 70 nsec clock cycle yields low power and robust timing margins.

Processor memory, PMEM, is implemented with 1Mbit DRAM's that are arranged in the cluster so that each PE has 16 Kbytes of data memory. A processor board has 16 Mbytes of memory, and a 16 board system has 256 Mbytes of memory. The MP-1 instruction set supports 32 bits of PE number and 32 bits of memory addressing per PE.

The MP-1 processor element (PE) design is different than that of a conventional processor because a PE is mostly data path logic and has no instruction fetch or decode logic. Like present RISC processors, each PE has a large on-chip register set (PREG) and all computations operate on the registers. Load and store instructions move data between the external memory (PMEM) and the register set. The register architecture substantially improves performance by reducing the need to reference external memory. The compilers optimize register usage to minimize load/store traffic.

Each PE has 40 32-bit registers available to the programmer and an additional 8 32-bit registers that are used internally to implement the MP-1 instruction set. With 32 PE's per die, the resulting 48 Kbits of register occupy about 30% of the die area, but represent 75% of the transistor count. Placing the registers on-chip yields an aggregate PE/PREG bandwidth of 117 gigabytes per second with 16K PE's. The registers are bit and byte addressable.

Each PE provides floating point operations on 32 and 64 bit IEEE or VAX format operands and integer operations on 1, 8, 16, 32, and 64 bit operands. The PE floating point/integer hardware has a 64-bit MANTISSA unit, a 16-bit EXPONENT unit, a 4-bit ALU, a 1-bit LOGIC unit, and a FLAGS unit; these units perform floating point, integer, and boolean operations. The floating point/integer unit uses more than half of the PS silicon area but provides substantially better performance than the bit-serial designs used in earlier massively parallel systems.

Most data movement within occurs on the internal PE 4-bit NIBBLE BUS and the BIT BUS (Figure 4). During a 32-bit or 64-bit floating point or integer instruction, the ACU microcode engine steps the PE's through a series of operations on successive 4-bit nibbles to generate the full precision result. Because the MP-1 instruction set focuses on conventional operand sizes of 8, 16, 32, and 64 bits, MasPar can implement subsequent PE's with smaller or larger ALU widths without changing the programmers instruction model. The internal 4-bit nature of the PE is not visible to the programmer, but does make the PE flexible enough to accommodate different front-end workstation data formats. The PE hardware supports both little-endian and big-endian format integers, VAX floating point F, D, and G formats, and IEEE single and double precision floating point formats.

UNIX Subsystem (USS): An important aspect of the system is the use of an existing computer system (specifically a VAX station 3520 *ULTRIX*^{TM} workstation) that follows existing industry standards (e.g. X windows, TCPIP, etc.). The USS provides a complete, network and graphic based, software environment in which all the MasPar tools and utilities (e.g. compilers) execute. Part of the application executes as a conventional workstation application; most of the "operating system" functions are provided by the workstation's UNIX software.

Communication Mechanism: The following sections describes the five major communications mechanisms.

1. **USS to ACU:** Three different types of interactions occur between USS and the ACU which use three different hardware support. All are based on a standard bus interface (VME). The following describes each mechanism:

   I. **Queues:** Hardware queues are provided which allows USS process to quickly interact with the process running on the ACU. The programming model is similar to UNIX pipes but with hardware assist.

   II. **Shared memory:** The shared memory mechanism overlaps ACU memory addresses with USS memory addresses. This provides a straitforward mechanism for processes to share common data structures like file control block etc.

   III. **DMA:** A DMA mechanism is provided that permits fast bulk data transfers without using programmed I/O.

2. **ACU to PE array:** Two basic capabilities are required for data movement between ACU and PE array: data distribution, DIST, array consensus detection which uses a global OR, GOR.

   I. **PE array: XNet** XNet communications provide all PE's with direct connection to its eight nearest neighbors. Processors on the physical edge of the array have toroidal wrapped edge connections [1][2].

      Three basic instruction types are provided to use the nearest neighbor connections [1][3]:

      a. **XNET:** The XNET instruction moves an operand from source to destination a specified distance in all active PE's. The instruction time is proportional to the distance times the operand size since all communication is done using single wire connections.

      b. **XNETP:** The XNETP instruction is pipelined so that a collection of PE's move an operand from source to destination over a specified distance. However the pattern of active and inactive PE's is very important since active PE's transmit data and inactive PE's act as pipeline stages. The instruction time is proportional to distance plus the operand size due to its pipelined nature.

      c. **XNETC:** The XNETC instruction is pipelined and is very similar to XNETP instruction except that a copy of the operand is left in all PE's acting as a pipeline stage. Again the instruction time is proportional to the distance plus the operand size.

   II. **PE array: Global Router** The global router is a circuit switched style network organized as a three stage hierarchy of crossbar switches. This mechanism provides direct point to point bidirectional communications. The network diameter is $\frac{1}{16}$ the number of PE's which requires a minimum of 16 communication cycles to do a permutation with all PE's. The basic instruction primitives are [1][3]:

      a. **ropen:** open a connection to a destination PE
      b. **rsend:** move data from the originator PE to the destination PE
      c. **rfetch:** move data from the destination PE to the originator PE
      d. **rclose:** terminate the communication

III. PE **array to I/O subsystem:** Since the global router provides high performance random PE to PE communication, the global router is also used to provide a high performance communication mechanism into the I/O subsystem. The interface is achieved by connecting the last stage of the global router to an I/O device, the I/O RAM. The programming model is identical to the model for using the global router.

3. **Array I/O system:** Refemng back to Figure 1, the I/O subsystem uses the following key components: the global router connection into the PE array (over 1 $\frac{GB}{sec}$), a large I/O RAM buffer (up to 256 MB), and a high speed (230 $\frac{MB}{sec}$) data communication channel between peripheral devices, a bus for device control (not for data movement). Using output as an example, the model for using the I/O subsystem follows these steps:

   a. Device is opened by the USS (all I/O devices are UNIX controlled)

   b. The ACU moves data into the I/O RAM through the global router.

   c. Either the USS or an I/O processor (IOP) schedules data movement from the I/O RAM to the device (e.g. Disk); data through the MPIOC and control on the VME bus.

   d. The USS is notified when the transaction is complete.

   Note that all transactions from the I/O Ram to external I/O systems can occur asynchronously from PE array actions. This is a key attribute since data can move into the I/O RAM at speeds over 1 $\frac{GB}{sec}$ then move at I/O device speeds, typically in the tens of megabytes per second or less, without affecting the performance of the PE array. These hardware mechanisms can support either typical synchronous UNIX I/O or newer (and faster) asynchronous software models.

3. THE SERIAL AND THE SIMD-BP ALGORITHMS

The parallel version of the backpropagation algorithm (referred to as the 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 [4] with output layer nonlinearities, and is further discussed in Section 5. Figure 5 shows the training procedures of the serial version of the backpropagation algorithm (BP) and its SIMD version (SIMD-BP).

To better describe the SIMD-BP training algorithm, we discuss the algorithm with the example of the 10-class Colorado problem, which 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. Figure 6 shows the PE array of MP-1 in a 128x128 grid array as it was arranged for this problem.

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 connection, the weights are changed. After that, the next pattern is presented to the network and this procedure is repeated for the new pattern. In SIMD version of this algorithm, the weights are not changed after each pattern. The weight changes are stored; after the completion of a sweep, they are added together and only then the weights are updated, based on the total weight change computed.

The following is the derivation of the backpropagation algorithm to clarify the difference between the SIMD-BP version and the sequential version.

Let us assume a network with N output neurons in a problem with P training patterns. The total squared error defined for one training sweep is defined as

$$E = \frac{1}{2P} \sum_{p=1}^{P} \sum_{n=1}^{N} (d_n^p - o_n^p)^2 \qquad (1)$$

Where d_n^p is the desired output value for the n^{th} output neuron for the p^{th} training pattern, and the o_n^p stands for the actual output of the n^{th} neuron for the p^{th} training pattern.

Below we first discuss the weight changes between the hidden and the output layers. Then, we describe the weight changes between the input and the hidden layer. The results can be easily generalized to more than one hidden layer. When there is no hidden layer, the first discussion is valid. Then, the hidden layer is the same as the input layer.

Using the chain rule we can find the rate of change of E with respect to w_{ij}, the weight connecting the j^{th} hidden neuron to the i^{th} output neuron, as

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_i^p} \times \frac{\partial o_i^p}{\partial w_{ij}}. \tag{2}$$

where

$$\frac{\partial E}{\partial o_i^p} = -\frac{1}{P} \sum_{p=1}^{P} (d_i^p - o_i^p)^2. \tag{3}$$

We assume a sigmoidal activation function in the form

$$o_i^p = \frac{1}{1 + e^{-\left[\sum_{j=1}^{M} x_j^p w_{ij} + \theta_i\right]}} \tag{4}$$

where M is the number of hidden neurons, and x_j^p is the j^{th} input to the output neuron, in other words, the output of the j^{th} hidden neuron. We get

$$\frac{\partial o_i^p}{\partial w_{ij}} = \frac{x_j^p e^{-\left[\sum_{j=1}^{M} x_j^p w_{ij} + \theta_i\right]}}{\left[1 + e^{-\left[\sum_{j=1}^{M} x_j^p w_{ij} + \theta_i\right]}\right]^2} = x_j^p o_i^p (1 - o_i^p). \tag{5}$$

Using Eqs. (3) and (6) in Eq. (2) gives

$$\frac{\partial E}{\partial w_{ij}} = -\frac{1}{P} \sum_{p=1}^{P} x_j^p o_i^p (1 - o_i^p)(d_i^p - of). \tag{6}$$

Therefore, using the gradient descent algorithm, the weight change for w_{ij}

$$\Delta w_{ij} = -\frac{\rho}{P} \sum_{p=1}^{P} x_j^p o_i^p (1 - o_i^p)(d_i^p - o_i^p). \tag{7}$$

where ρ is a small constant called the step size.

For the weights connecting the input layer to the hidden layer, the derivation is slightly more complicated. Let us assume that v_{jk} is the weight connecting the k^{th} input neuron to the j^{th} hidden neuron. Then, we have

$$E = \frac{1}{2P}\sum_{p=1}^{P}\sum_{n=1}^{N}(d_n^p - o_n^p)^2 = \frac{1}{2P}\sum_{p=1}^{P}\sum_{n=1}^{N}\left[d_n^p - \frac{1}{1 + e^{-\left[\sum_{j=1}^{M} x_j^p w_{ij} + \theta_i\right]}}\right]^2 . \tag{8}$$

where x_j^p is the output of the j^{th} hidden neuron for the p^{th} training pattern and is given by

$$x_j^p = \frac{1}{1 + e^{-\left[\sum_{k=1}^{K} i_k^p v_{jk} + \theta_j\right]}} . \tag{9}$$

where K is the number of input neurons (ie. the length of the input pattern), and i_k^p is the k^{th} bit* of the p^{th} training pattern. Using the chain rule again, we get

$$\frac{\partial E}{\partial v_{jk}} = \frac{\partial E}{\partial x_j^p}\frac{\partial x_j^p}{\partial v_{jk}} . \tag{10}$$

Using

$$\frac{\partial E}{\partial x_j^p} = \frac{1}{P}\sum_{p=1}^{P}\sum_{n=1}^{N}(d_n^p - o_n^p)\frac{-w_{nj}e^{-\left[\sum_{j=1}^{M} x_j w_{nj} + \theta_n\right]}}{\left[1 + e^{-\left[\sum_{j=1}^{M} x_j w_{nj} + \theta_n\right]}\right]^2} = \frac{-1}{P}\sum_{p=1}^{P}\sum_{n=1}^{N} w_{nj} o_n^p (1 - o_n^p)(d_n^p - o_n^p),$$

and

$$\frac{\partial x_j^p}{\partial v_{jk}} = i_k^p x_j^p (1 - x_j^p), \tag{11}$$

we get

* In binary representation of the input pattern, the k^{th} bit has a value of 1 or 0, whereas in continuos number representation, this input is the k^{th} component on the analog input pattern vector.

$$\frac{\partial E}{\partial v_{jk}} = -\frac{1}{P}\sum_{p=1}^{P} i_k^p x_j^p(1 - x_j^p)\sum_{n=1}^{N} w_{nj} o_n^p(1 - o_n^p)(d_n^p - o_n^p). \tag{12}$$

The weight change for steepest descent is

$$\Delta v_{jk} = \frac{\rho}{P}\sum_{p=1}^{P} i_k^p x_j^p(1 - x_j^p)\sum_{n=1}^{N} w_{nj} o_n^p(1 - o_n^p)(d_n^p - o_n^p). \tag{13}$$

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, the weight update is performed after each training pattern in the serial implementation. In other words, using (7) and (13), the weight changes are computed as

$$\Delta w_{ij} = \rho x_j^p o_i^p(1 - o_i^p)(d_i^p - o_i^p). \tag{14}$$

and

$$\Delta v_{jk} = \rho i_k^p x_j^p(1 - x_j^p)\sum_{n=1}^{N} w_{nj} o_n^p(1 - o_n^p)(d_n^p - o_n^p). \tag{15}$$

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 proven to work, its speed is not only slow, but the minimum that it reaches might also be a different minimum than the minimum the exact algorithm would have found. Figure 7 shows the descent steps taken to move to the minimum of a paraboloid by the exact algorithm and the approximate version.

The SIMD-BP, however, 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*. Then, the weight vectors on all the networks are updated based on the weight change vector. This vector is sent to all PEs of MP-1 using the XNET structure.

The use of the exact algorithm results in data parallelism, and most of the speed-up achieved is due to this type of parallelism. It is also theoretically more accurate. Thus there are two different types of parallelism exploited in the SIMD-BP as follows:

- **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 each stage can be performed all at the same time. Since there are no connections between the neurons of the same stage, no communication overhead is necessary*.

Figure 6 shows the architectural parallelism for the Colorado data set. Each network is simulated by 100 PEs, which is the size of the hidden layer of the backpropagation network. The total network for the 10-class Colorado set consisted of 7 input neurons, 100 hidden neurons, and 10 output neurons.

- **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 parallel. Therefore we can now simulate more than one network at the same time. They all have the same initial random weights and ideally one input pattern to learn. These input patterns, however, are different from one network to another. Each network calculates weight changes 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 the weights of all the networks are updated, simultaneously based on the accumulated weight changes from all the networks. 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 training patterns in the training set. For example the 10-class Colorado problem has 1188 patterns in its training set and the number of PE's required for each network is 100. Therefore the maximum number of networks running simultaneously is $\frac{16384}{100} = 163$. For simplicity, we chose to have: only 156 networks running simultaneously**.

  94 networks were given 8 patterns and the remaining 62 were given 7 patterns ($7x62 + 8x94 = 1188$), which gives a degree of virtualization of 8. Hence, we are computing the weight changes for 156 patterns each clock cycle. Figure 6 shows the layout of the 156 networks in the MasPar PE array.

In any parallel machine, the degree of parallelism is limited to the physical parallel resources of the machine. For example, in the 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 a lot 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 pattern, we ideally would have required

* One could assign a PE to every neuron in the network. However, this does not bring a higher degree of parallelism than the case when there is only *w* many PEs assigned lo the network as the number of neurons m the largest layer. This is due to the serial nature of the stages with respect to each other and the communication overhead required for communication between two layers.

** If we had chosen 163 networks running simultaneously, loading the input patterns into the PEs correctly would become mom difficult and the communication pattern among the PEs would have become irregular which would have caused the PE-to-PE communication to be achieved in several serial steps rather than one parallel step

100x1188 = 118800 PEs. Since this many PEs were not available, we implemented a concept referred to as virtualization. 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 (Figure 8). 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.

The data distribution among the PEs has to take this into account. Each PE receives the data for all the virtual PEs which it is going to emulate on all the virtual layers. Care must be taken in loading the data into the PEs, so that each PE receives only the data which the virtual PEs it is assigned to would have received. Also the programmer must be careful about the fact that in the last slice there might not be enough data to require the services of the entire PE array. In this case, those PEs which have run out of data, must be inactive for the computations of the last slice.

Figure 5 shows the block diagram of the serial backpropagation and the SIMD version of backpropagation. The MP-1 program is designed to arrange the PE array to achieve the minimum degree of virtualization and 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 two parameters: 1.The size of the hidden layer of the network. 2.The number of training patterns. For example, for a classification problem with 500 training patterns and a network with the hidden layer of 20 neurons, the program requires no virtualization (virtualization degree of zero). Figure 9 shows the PE array arrangement for this problem. The remaining part of the MP-1 takes the degree of virtualization 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 that slice. The program then performs the operations of each slice separately. It first deactivates the PEs not required for that slice and then has the ACU decode the instructions and send them to the PEs, which in turn perform the operation if their enable flag is high. The MP-1 program. thereby, is written in a way that it detects and adjusts to the size of any given problem automatically.

Figure 6 shows how the networks are organized in the MP-1 implementation in the Colorado problem. The first 128 networks were chosen in a vertical layout fashion and the remaining 28 in the horizontal layout fashion. This produces the simplest communication pattern. An inverse layout pattern (first 128 horizontal and the rest vertical), would result in additional communication overhead to distribute the input patterns to all the PEs in each network. Further speed-up can be achieved by assigning 10 x 10 square of PEs to each network instead of a 1 x 100 array of PEs. At the cost of a more complicated communication pattern, this could result in a slight speed-up.

The way the networks are organized is such that the first PE in the all networks can easily be enabled. The input patterns are loaded into the first PEs of the networks using the

parallel read command [1]:

```
cc = p_read(d, buf, nbytes)
plural int cc;
int d;
plural char *buf;
int nbytes;
```

This command was used in the following format:

```
if ( (iyproc==0) || ((iyproc>=hn)&&(ixproc==0)) )
   Fstatus=p_read(fd, &x[slice][0], invecbt);
```

The if statement enables the first PE of each network (Figure *6)*. *ixproc* and *iyproc* are the x and the y coordinates of each PE, respectively, in the 128x128 PE array. ***hn*** is the size of the hidden layer (in this case 100). ***invecbt*** is the size of the input vector in bytes, and ***slice*** is the degree of ***virtualization.*** Notice that the entire input vector is read into the first PE in one shot.

After the loading of input data, The first PEs proceed to communicate the data to the rest of the PEs in their networks. This communication uses the ***xnetc*** command [1]. The xnetc command was used as follows:

```
if (iyproc==0)
   xnetcS[hn-1].x[slice][i] = x[slice][i];

if ( (ixproc==0) && (iyproc >= hn) )
   xnetcE[hn-1].x[slice][i] = x[slice][i];
```

The if statements enable the first PEs of the networks. The letters "*S*" and "*E*" specify the direction in which data should be sent (South and East). ***hn-1*** is the step size, which means send $100-1=99$ PEs to the south or east. Notice that since ***xnetc*** is used, a copy of the communicated data is left in each relaying PE memory at the right location.

The forward calculation of data also requires some communication which uses ***xnetp*** and ***xnetc.*** To calculate the total AW (the change in the weight matrix), we used two library routines from MP-1's mathematics library MPML [1]. These two routines are:

```
void fp_matsumtovex ( ny, nx, B, nxB, yoffB, xoffB, VX )

int ny, nx, nxB, yoffB, xoffB;

plural float *B, *VX;
```

and

```
void fp_matsumtovey ( ny. nx, B, nxB, yoffB, xoffB, VY )

int ny, nx, nxB, yoffB, xoffB;

plural float *B, *VY;
```

The first routine adds the columns of the matrix B from the row *yoffB* and the column *xoffB* for *ny* rows and *nx* columns and puts the results in the x-oriented vector *VX*. The second routine adds the rows of this submatrix and puts in the y-oriented VY vector.

For example, one could use the fp_matsumtovey library routine to add the processor numbers (iproc*) assigned to each processor row by row from the 4th row to the 100th row, from the 6th PE in each row through the 120th PE in that row and put the sum values in a Y-oriented vector in the 0th column of the PE array. The steps to perform this operation are as follows:

```
1  plural float B. VY;

2  B = (plural float) iproc;

3  fp_matsumtovey( 96 , 114. @B , 1 , 3 , 5 , @VY );
```

In statement 1, the variables B and VY are declared across all processors. In statement 2, the iproc value of each PE is assigned to the variable B of that PE. In statement 3, the fp_matsumtovey function is used to add the values of the B variables in each row from the 4th to the 100th row, and each row from the 6th element to the 120th element and put the result of each row in the VY variable of the first PE of that row** (see Figure 10).

The backward propagation of error and updating the weights uses the same routines in the reverse direction of the network.

* In the PE array of MP-1 each PE can be identified in two ways. First way is to identify the row number *ixproc* and the column number *iyproc* of the PE in the two dimensional PE grid array. The second way is to identify the processor number *iproc* of the PE (see Figure 10). Where $iproc=ixproc*nxproc+iyproc+1$ and $nxproc$ it the number of PEs in a row(in 16K machine, 128). Therefore the expressions *proc[3][4].B* and *proc[389].B* are equivalent and both point to the value of the variable B of the PE in the 4th row and the 5th column.

** The number of PEs in the Y direction $ny=100-4=96$

the number of PE's in the X direction $nx=120-6=114$

The starting row $yoffB=4-1=3$; he first PE in each row is he 0th PE

The starting PE number in every row $xoffB=6-1=5$; he first PE in each row is he 0th PE

4. TIME COMPLEXITY ANALYSIS

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

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

The Serial BP algorithm

Let us denote the number of input neurons to the network with p, the number of hidden neurons with n_h (assuming one hidden layer in the network), the number of output neurons with n_o, and the number of training patterns in the training set with k. Since, in the first stage, a backpropagation network has to perform one multiplication for every connection, we get p $\times n_h$ floating point multiplications for the first stage. To add the incoming signals to each neuron and subtract the result from a threshold, we need $n_h \times p$ floating point additions for the first stage. In the same way, we can find $n_h \times n_o$ floating point multiplications, and $n_o \times n_h$ floating point additions for the second stage. Therefore we get a total of $n_h \times \left[p + n_o\right]$ floating point multiplications, and $n_h \times \left[p + n_o\right]$ floating point additions.

Let us denote the time required for a floating point addition by α and the time needed for a floating point multiplication by μ. Since the error backpropagation through the net and weight changes require the same order of floating point additions and floating point multiplications as forward propagation, and since this procedure is repeated k times, once for each pattern, the time complexity of the backpropagation network becomes

$$T_{BP} = \theta \left[k \times n_h \times \left[p + n_o\right] \times \left[\alpha + \mu\right]\right] \tag{16}$$

The SIMD-BP algorithm

To calculate the time complexity of the SIMD-backpropagation, in addition to the time required for floating point additions and multiplication, we have to consider the communication overhead. Let us first consider the additions and the multiplications. Since in SIMD-BP all the neurons of each stage operate in parallel, we only need p multiplications and p additions for the first stage and n_h multiplications and n_h additions for the second stage. Thus, the computation time for the process is on the order of $\left[p + n_h\right] \times \left[\alpha + \mu\right]$. 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 $nyproc \times C$, where C is the time it takes to communicate a float value from one PE to its immediate neighbor, and $nyproc$ is the length of the PE array in the y direction ($nyproc$=128).

Thus, we get

$$T_{SIMD-BP} = \theta\left[\left[\left(p + n_h\right) \times \left[a + \mu\right] + \mathrm{nyproc} \times C\right] \times slice\right] \tag{17}$$

where slice is the degree of virtualization.

The order of estimated speed-up is to be measured by $\frac{T_{BP}}{T_{SIMD-BP}}$. Equations (16) and (17) give

$$\frac{T_{BP}}{T_{SIMD-BP}} = \theta\left[\frac{k \times n_h \times \left(p + n_o\right) \times \left(\alpha + \mu\right)}{\left[\left(p + n_h\right) \times \left(\alpha + \mu\right) + nyproc \times C\right] \times slice}\right] \tag{18}$$

For example, in the 10-class remote sensing problem, we have: p = 7, $k = 1188$, $n_h = 100$, $n_o = 10$, slice = 8. Thus,

$$\frac{k \times n_h \times \left(p + n_o\right) \times \left(\alpha + \mu\right)}{\left[\left(p + n_h\right) \times \left(\alpha + \mu\right) + \mathrm{nyproc} \times C\right] \mathrm{xslice}} = \frac{1188 \times 100 \times \left(7 + 10\right) \times \left(\alpha + \mu\right)}{\left[\left(7 + 100\right) \times \left(\alpha + \mu\right) + 128 \times C\right] \times 8}$$

$$= \frac{2019600 \times \left(\alpha + \mu\right)}{856 \times \left(\alpha + \mu\right) + 1024 \times C}$$

Since MasPar PE's are 4 bit processors, we can assume that a 32 bit floating point addition takes 4 clock cycles. Furthermore, let us assume that a floating point multiplication takes twice as long as a floating point addition, namely 8 clock cycles, and that each communication cycle to a neighboring PE using XNet requires 4 clock cycles. With the above assumptions, the ratio given by Eq. (18) becomes

$$= \frac{2019600 \times \left(4 + 8\right)}{856 \times \left(4 + 8\right) + 1024 \times 4} = \frac{24235200}{11300} = 2144.7$$

In our experiments with backpropagation on a Sun 3/60 station, each sweep of training for the 10-class problem takes an average of approximately 7 minutes and 30 seconds. On MasPar, on the other hand, every 100 sweeps takes an average of approximately 14 seconds. This results in a speed-up factor in this particular case equal to

$$\frac{\left(7\times 60+30\right)\times 100}{14}=3214$$

Figure 11 shows the error curves of different SIMD-BP networks run for the two-stage network. As shown in this figure the error decay is a smoothly exponentially decaying function which is the characteristic of the exact algorithm. The error function of the serial network is only piecewise exponentially decaying. Figure 12 shows the: run times for different size hidden layers of the SIMD-BP.

5. THE SIMD DELTA RULE ALGORITHM

In a number of applications, it is sometimes preferred to remove the hidden layer(s). Then, there are just the input and the output layers. The derivations of the Equations (1) through (7) still apply. The error function is defined as in (1) and the gradient descent algorithm results in the weight change of

$$\Delta w_{ij} = -\frac{\text{P}}{P}\sum_{p=1}^{P} x_j^p o_i^p (1 - o_i^p)(d_i^p - \text{of}). \tag{19}$$

as before. Since there are no hidden layers, this weight change equation applies to all the weights in the network. The backpropagation algorithm for two-layer networks is also called the *Delta rule* algorithm [4].

Since there is no hidden layer in the two-layer network, the number of PE's assigned to each network on the MP-1 PE grid depends on the number of neurons in the output layer of the network. This is determined by the coding scheme used for output.

The time complexity of such a network is as follows.

The serial delta rule algorithm

As before we denote p to be the number of input neurons, n_o the number of output neurons, and k the number of training patterns in the training set. Since there are two layers of neurons, there is only one stage of connections between the layers. In this stage, the *Delta rule* performs one multiplication for every connection, hence p x n_o floating point multiplications, and p x n_o floating point additions to add the incoming signals to the output neurons and subtract them from a threshold.

If, as before, we denote the time required to perform a floating point addition and a floating point multiplication by a and β, respectively, the time complexity of the serial backpropagation network can be estimated as

$$T_{BP} = \theta\left[k \times p \times n_o \times \left[\alpha + \beta\right]\right] \tag{20}$$

The SIMD delta rule algorithm

Similar to the case of networks with hidden layers, in addition to the time required for floating point addition and multiplication, the communication overhead also has to be taken into account in the parallel algorithm. For this purpose, as before, the value C is introduced as the time required for a floating point value to be sent from a PE to its immediate neighbor.

Since the operations in the stage are performed in parallel, there are only p floating point multiplications and p floating point additions. Thus the total time required for all the additions and the multiplications is p x $[\alpha+\beta]$. Since the PE array is *nxproc* x nyproc, which is 128 x 128 in the 16K machine, the communication overhead is at most on the order of C x nyproc. Therefore, the time complexity can be estimated as

$$T_{SIMD-BP} = \theta\left|\text{slice x}\left[p \times \left[\alpha+\beta\right] + C \times nyproc\right]\right| \tag{21}$$

where slice, is as, before the degree of virtualization. Hence, the theoretical speed-up factor can be estimated as

$$\frac{T_{BP}}{T_{SIMD-BP}} = \theta\left[\frac{k \times p \times n_o \times \left[\alpha+\beta\right]}{slice \times \left[p \times \left[\alpha+\beta\right] + C \times nyproc\right]}\right] \tag{22}$$

6. CONCLUSIONS

Implementing neural network algorithms in massively parallel machines is very promising to reduce the implementation time from hours to minutes. This kind of speed-up is impossible to achieve with a serially fast neural network algorithm.

The backpropagation algorithm has architectural parallelism and data parallelism in the way it is parallelized in this article. While architectural parallelism is limited by the size of the layers of the network, the data parallelism is only limited by the number of PEs available and the number of training patterns, which is often far more than the number of neurons in a layer.

Massively parallel implementations of neural networks allow larger problems to be investigated in a short amount 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.

REFERENCES

[1] MasPar MP-1 Reference Manuals, MasPar Computer Corporation, Sunnyvale, CA

[2] Kenneth E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transaction on Computers,* Vol. C-29, pp. 836-840, Sept. 1980.

[3] Peter Christy, "Software To Support Massively Parallel Computing on the MasPar MP-1"; *Proceedings of the IEEE Compcon Spring* 1990, Feb. 1990.

[4] D.E. Rumelhart, J.L. McClelland, *Parallel Distributed Processing,* The MIT press, Cambridge Massachusets, 1986.

[5] P.J. Werbos, "Backpropagation: Past and Future", *Proceedings of ICNN* 88, San Diego, Cal., pp.343-353, June 1988.

[6] 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.

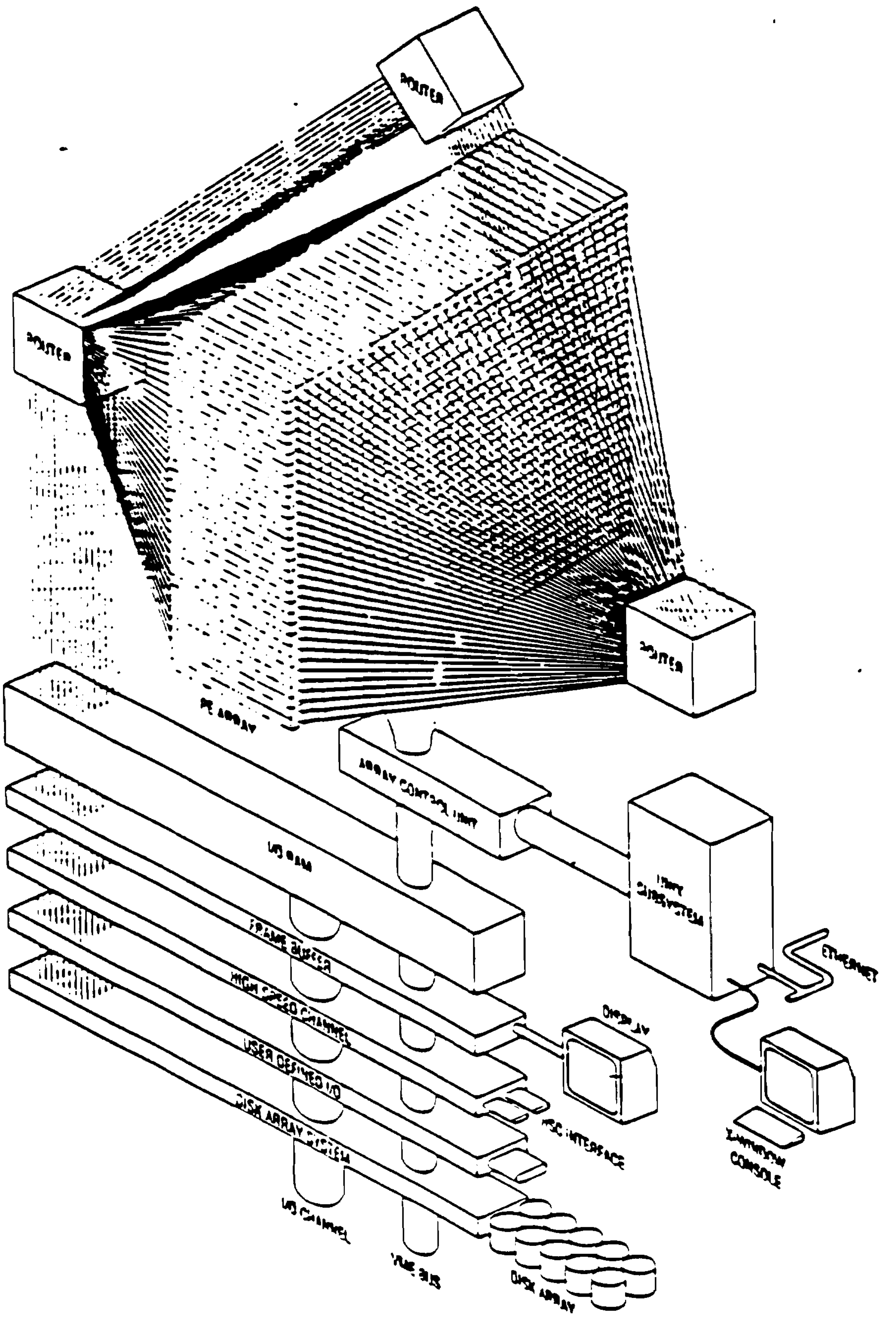


Figure 1. Block diagram of MasPar MP-1.

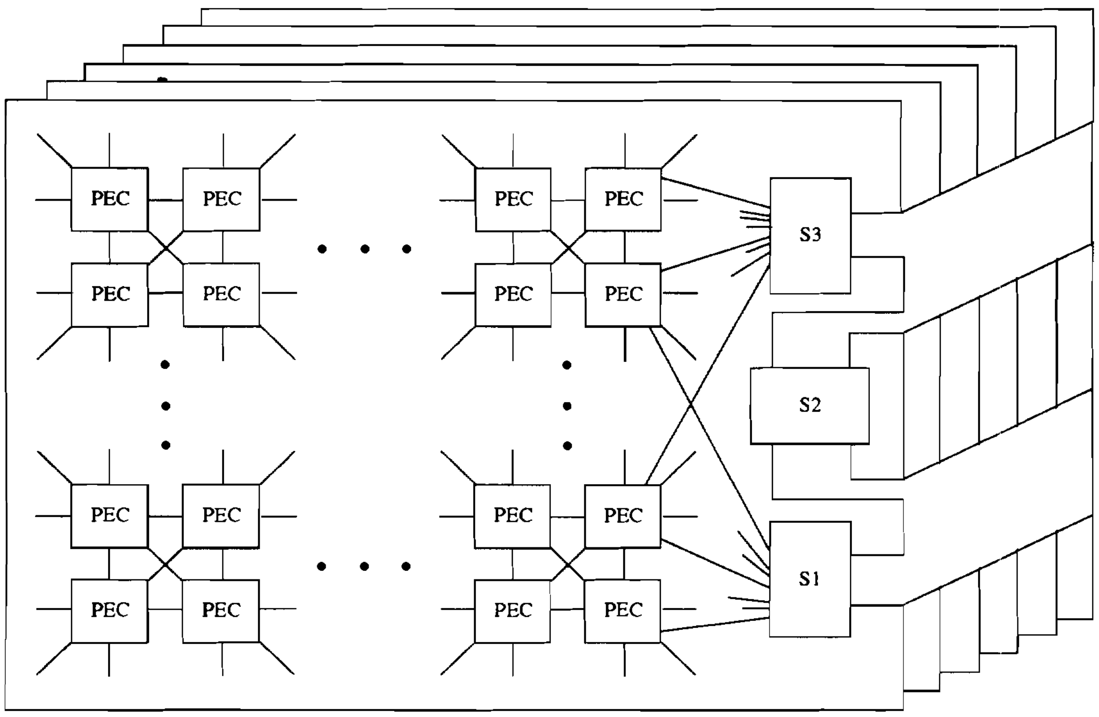


Figure 2. Block diagram of array processor of MasPar.

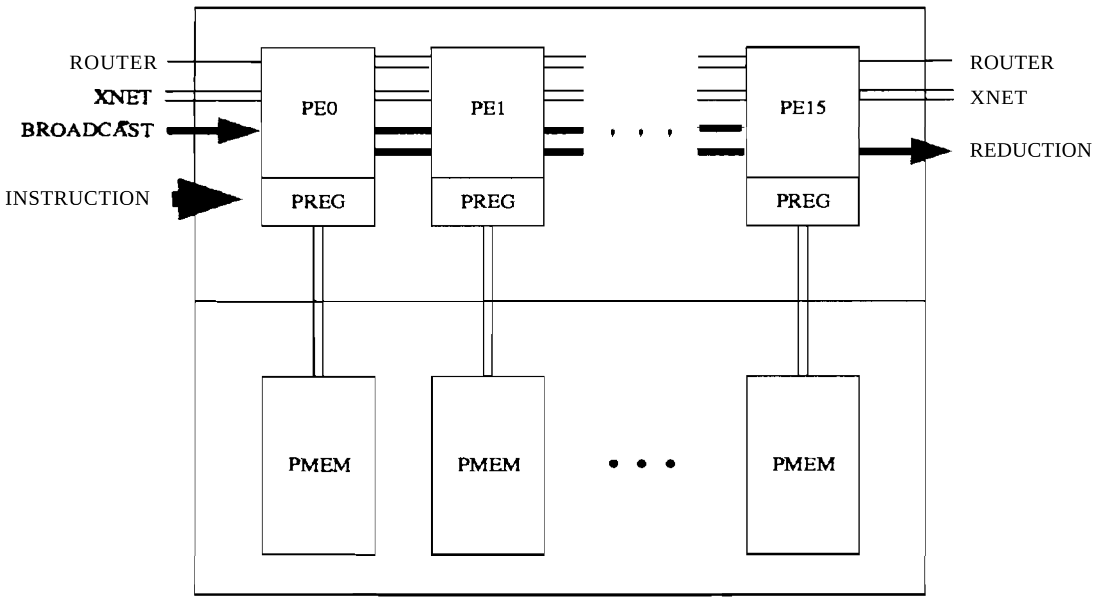


Figure 3. Block diagram of a PE cluster of MasPar

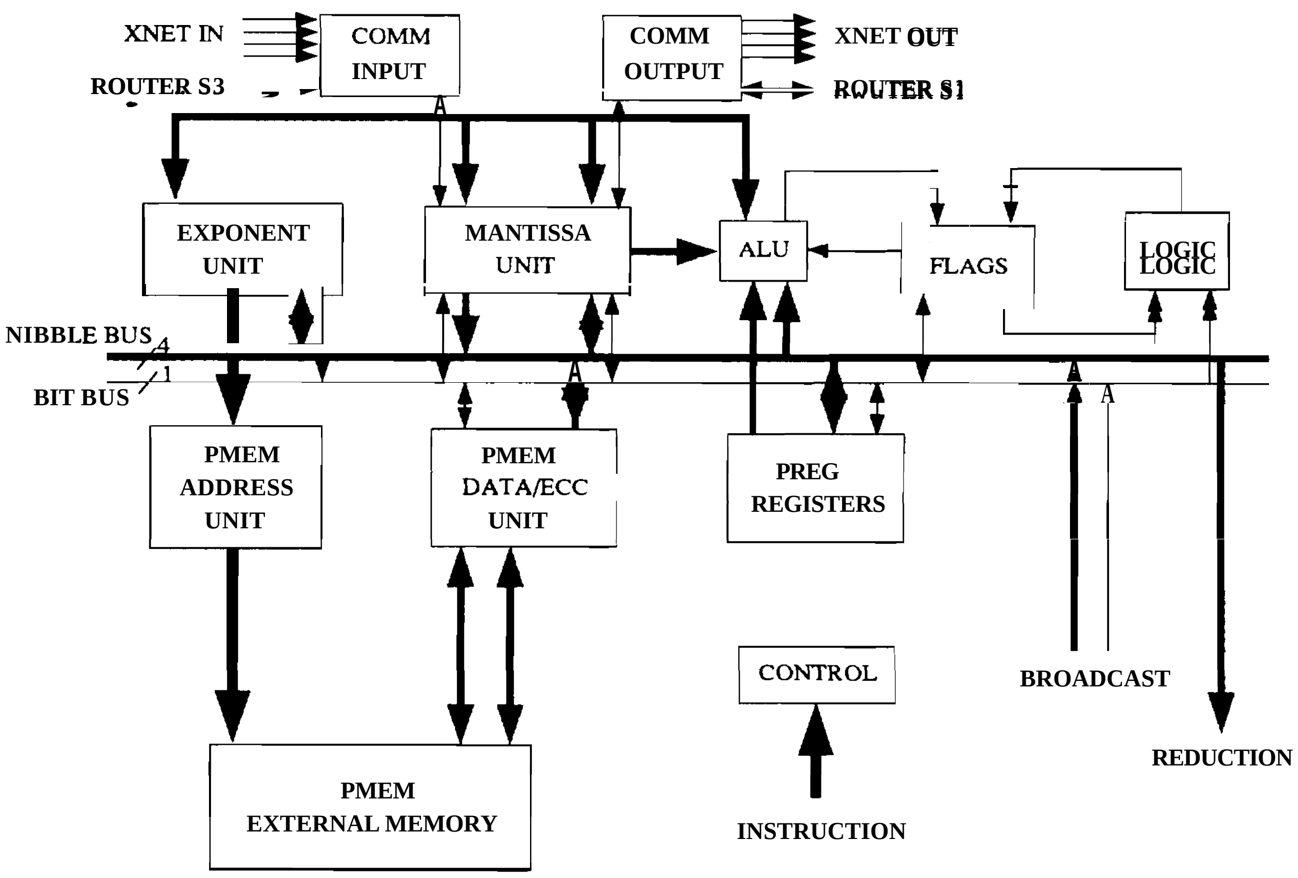
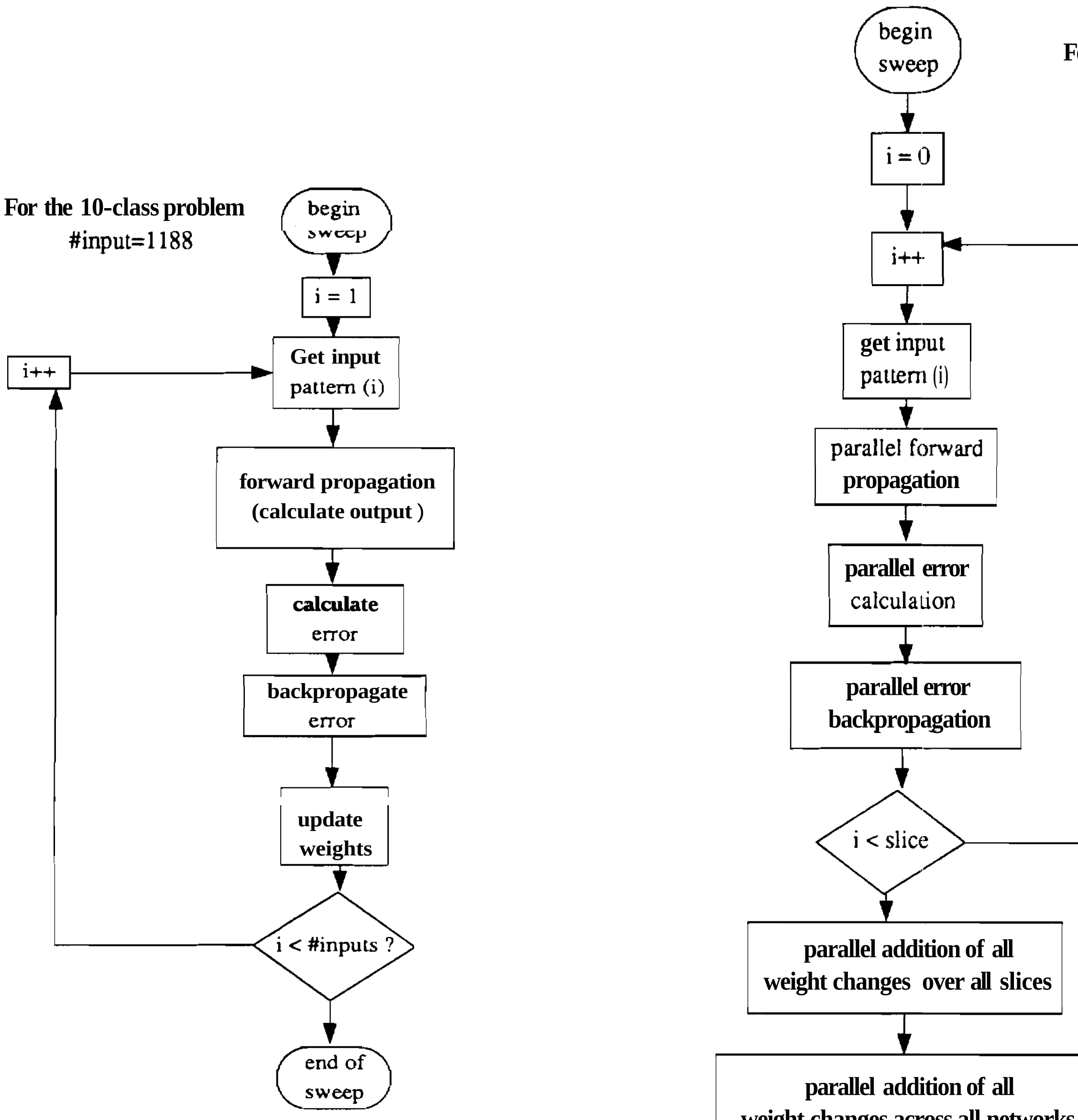
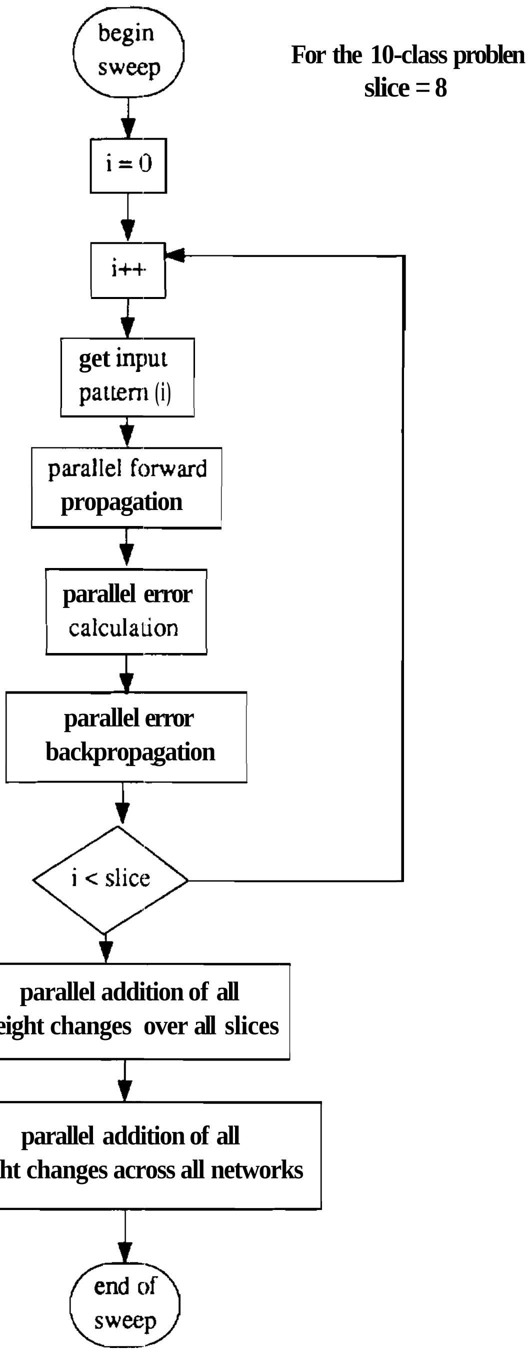


Figure 4. Internal architecture of a PE.




For the 10-class problen
slice = 8
begin sweep
i = 0
i++
get input pattern (i)
parallel forward propagation
parallel error calculation
parallel error backpropagation
i < slice
parallel addition of all weight changes over all slices
parallel addition of all weight changes across all networks
end of sweep

(a) (b)

Figure 5. Flow chart of (a) serial BP and (b) SIMD-BP.

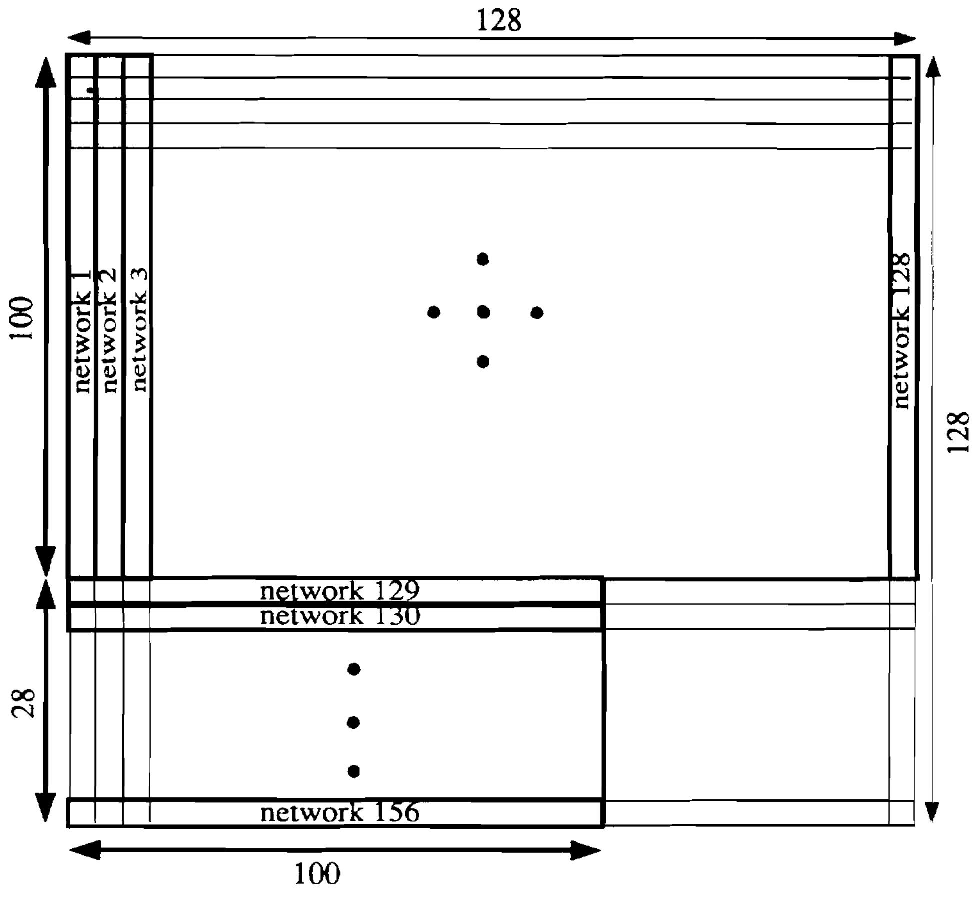


Figure 6. PE array of MasPar partitioned for the Colorado data set.

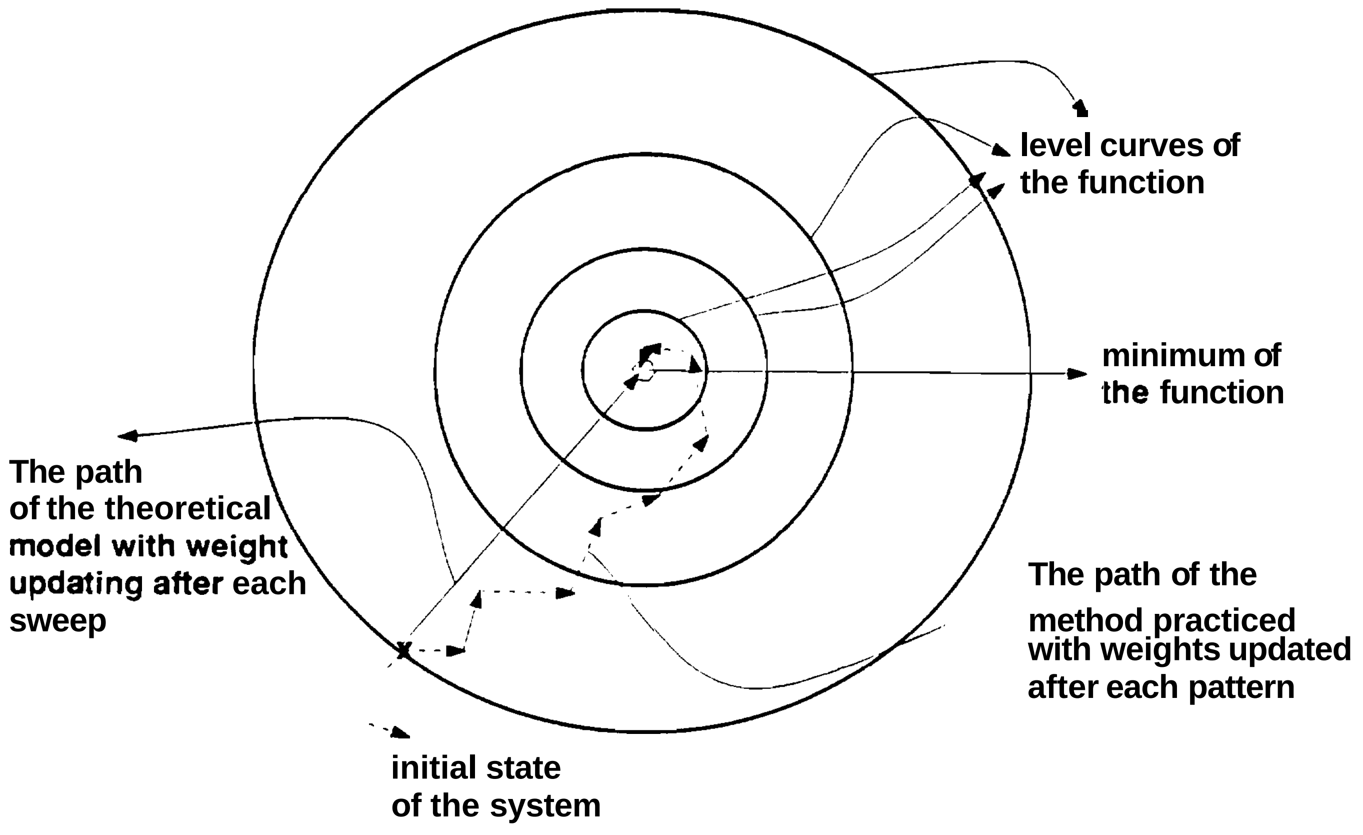

level curves of the function
minimum of the function
The path of the theoretical model with weight updating after each sweep
The path of the method practiced with weights updated after each pattern
initial state of the system

Figure 7. The descent path toward the minimum of a paraboloid function for updating the weights after each pattern and after each sweep.

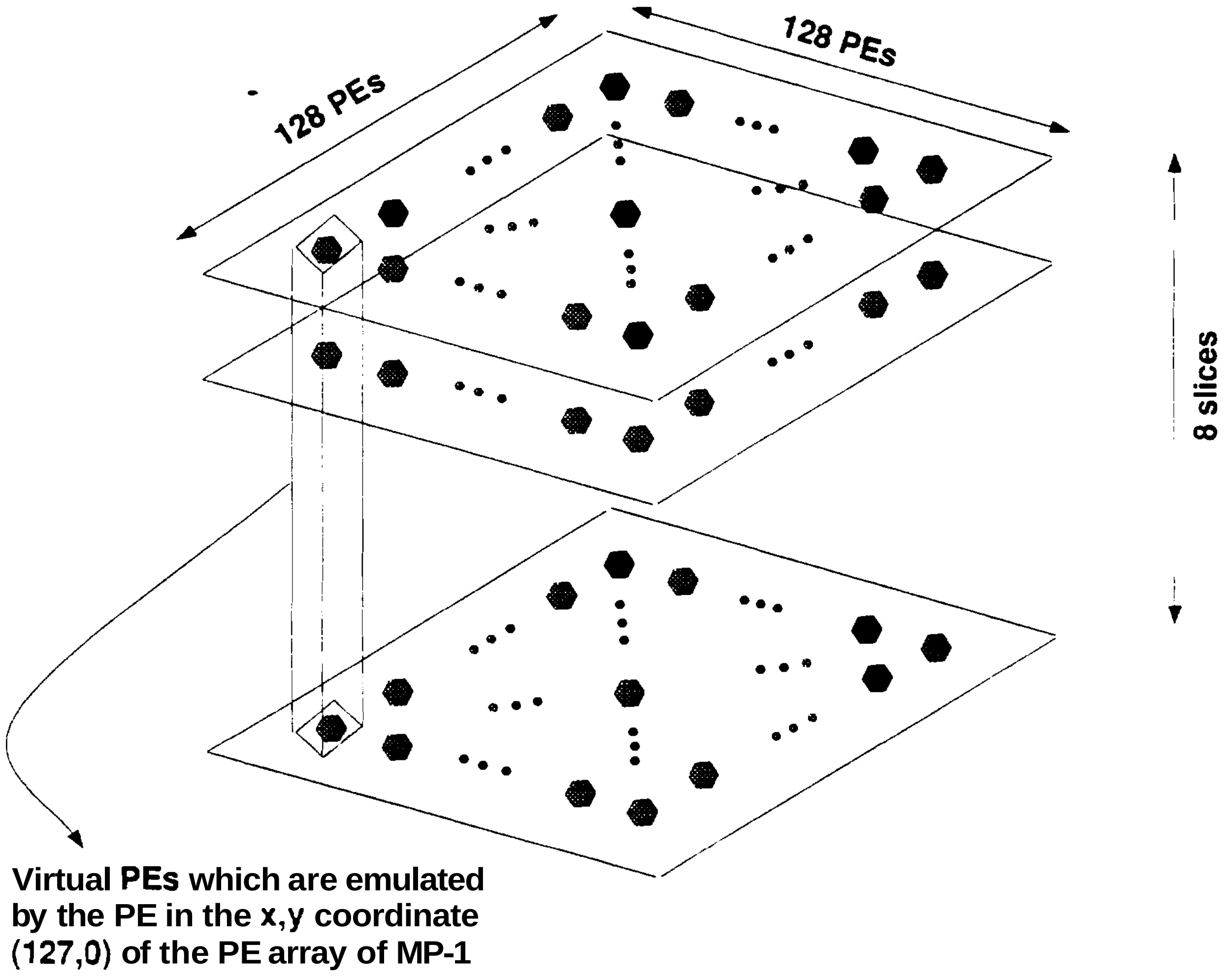

128 PEs
128 PEs
8 slices
Virtual PEs which are emulated by the PE in the x,y coordinate (127,0) of the PE array of MP-1

Figure 8. **The 3-D virtual array for the 10-class Colorado data set.**

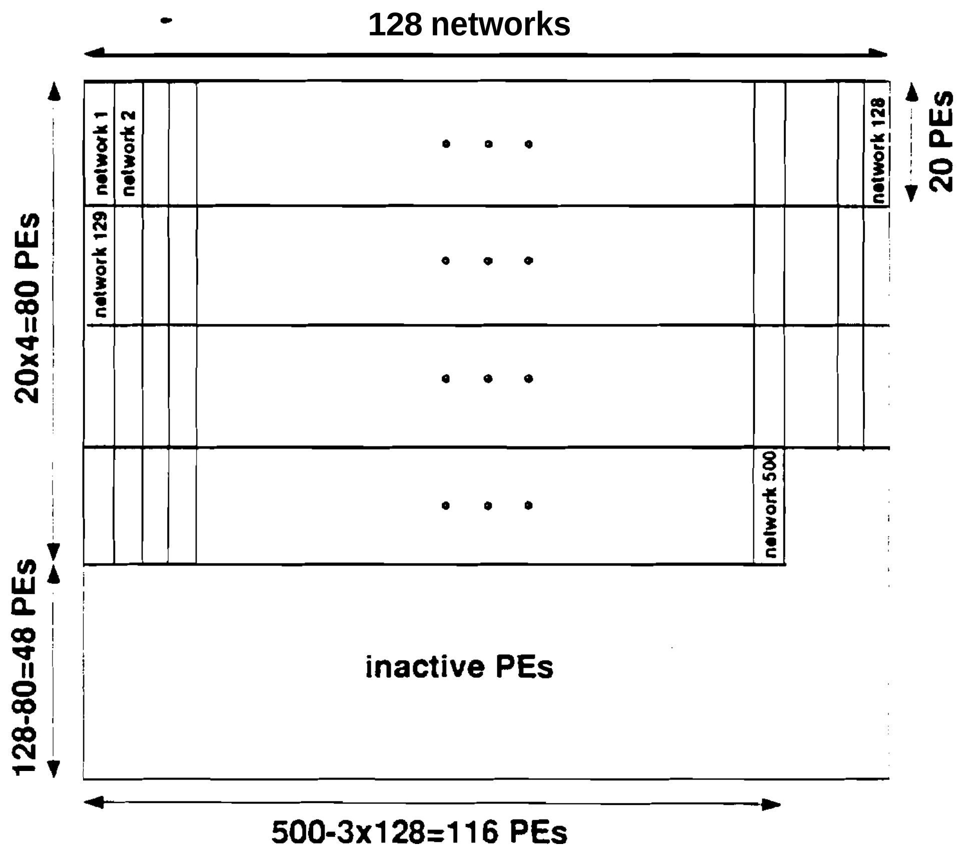


Figure 9. The PE arrangement for hidden layer size of 20 and training set size of 500.

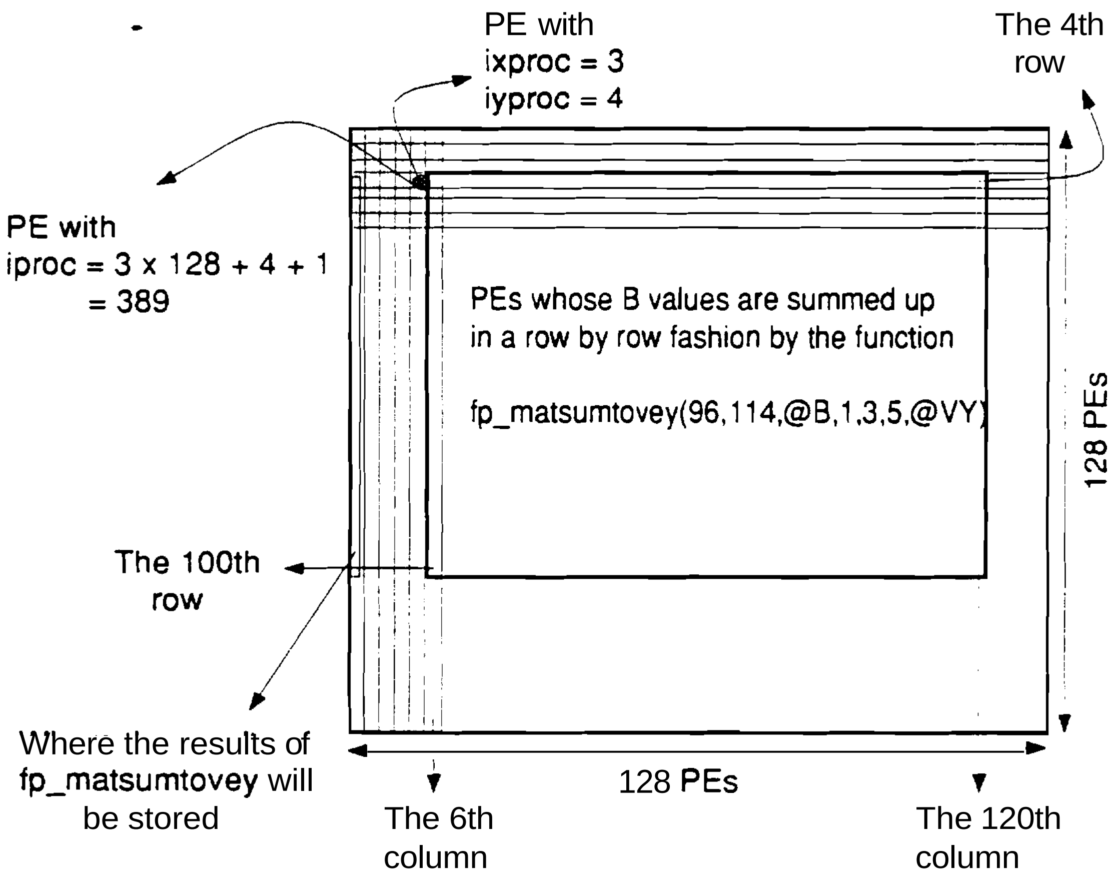

PE with
ixproc = 3
iyproc = 4
The 4th row
PE with
iproc = 3 x 128 + 4 + 1
= 389
PEs whose B values are summed up in a row by row fashion by the function
fp_matsumtovey(96,114,@B,1,3,5,@VY)
128 PEs
The 100th row
Where the results of fp_matsumtovey will be stored
128 PEs
The 6th column
The 120th column

Figure **10. An example of the operation of the** fp_matsumtovey **routine.**

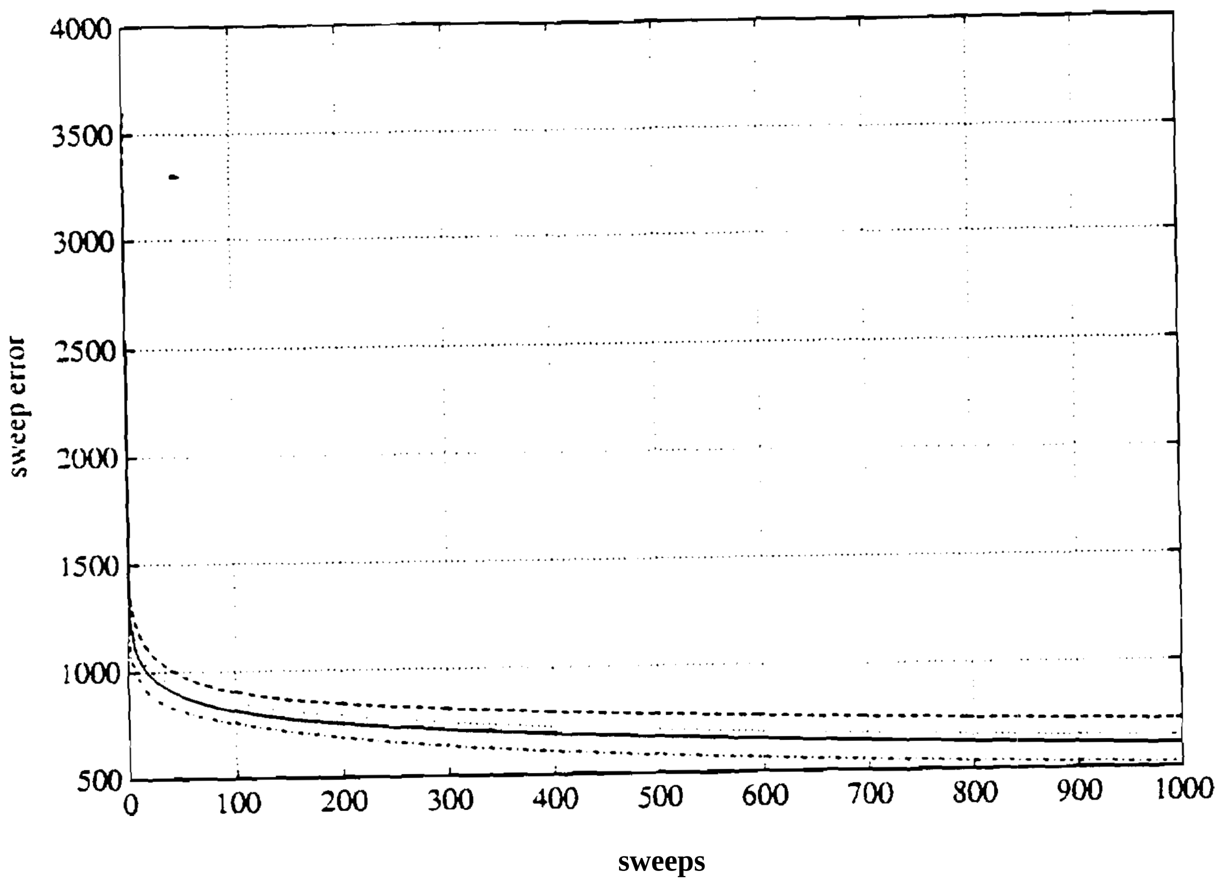


Figure 11. Error curves of SIMD-BP: top to bottom. 90. 110, 100, 120 hidden neurons.

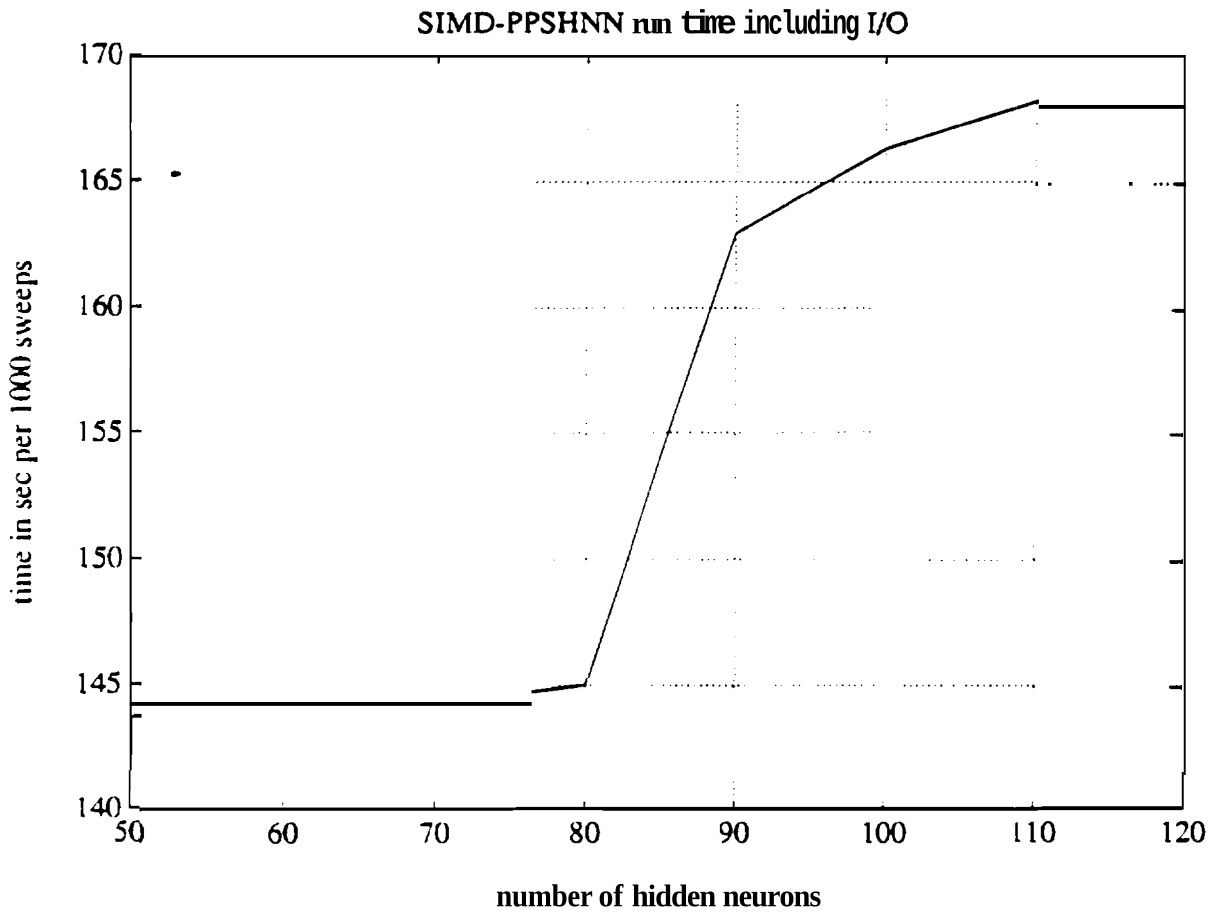


Figure 12. SIMD-BP run times.