JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 14, 248–259 (1992)

The Ring Array Processor: A Multiprocessing Peripheral for Connectionist Applications

NELSON MORGAN, JAMES BECK, PHIL KOHN, JEFF BILMES, ERIC ALLMAN, AND JOACHIM BEER

International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, California 94704-1105

We have designed and implemented a Ring Array Processor (RAP) for fast implementation of our continuous speech recognition training algorithms, which are currently dominated by layered "neural" network calculations. The RAP is a multi-DSP system with a low-latency ring interconnection scheme using programmable gate array technology and a significant amount of local memory per node (4–16 Mbytes of dynamic memory and 256 Kbytes of fast static RAM). Theoretical peak performance is 128 MFLOPS/board. A working system with 20 nodes has been used for our research at rates of 200–300 million connections per second for probability evaluation, and at roughly 30–60 million connection updates per second for training. A fully functional system with 40 nodes has also been benchmarked at roughly twice these rates. While practical considerations such as workstation address space restrict current implementations to 64 nodes, the architecture scales to about 16,000 nodes. For problems with 2 units per processor, communication and control overhead would reduce peak performance on the error back-propagation algorithm to about 50% of a linear speedup. This report describes the motivation for the RAP and shows how the architecture matches the target algorithm. We further describe some of the key features of the hardware and software design. © 1992 Academic Press, Inc.

INTRODUCTION

We have designed and implemented a Ring Array Processor (RAP) for fast implementation of layered "neural" network calculations. The RAP is a scalable multi-digital signal processor (DSP) system with a low-latency ring interconnection scheme using programmable gate array technology and a significant amount of local memory per node. This report describes the motivation for the RAP and shows how the architecture matches the target algorithms. We further describe some of the key features of the hardware and software design.

In our speech recognition research, we have been experimenting with layered "neural" algorithms as probabilistic estimators for a Hidden Markov Model (HMM) procedure [5, 6, 19]. Features representing the spectral content of the speech are estimated 100 times per second. A layered network is trained to predict the phonetic label of each 10-ms "frame." This network takes as its input the spectral features from one or more frames, and has an output layer consisting of one unit per phonetic category. For some of our experiments, the inputs are real-valued, and hidden layers are used. For others, the speech features are vector-quantized to map the frame into one of a set of prototype vectors, and the network input consists of a binary input unit for each possible feature value, only one of which can be active at a time. In either case, the neural network is trained by back-propagation [26, 27], augmented by a generalization-based stopping criterion [18]. It can be shown [7] that the net outputs can be trained to estimate emission probabilities for the Viterbi decoding step of an HMM speech recognizer. A network is useful for this procedure because it can estimate joint probabilities (joint over multiple features or time frames) without strong assumptions about the independence or parametric relation of the separate dimensions. We have conducted a number of experiments which seem to confirm the utility of this approach.

For continuous speech recognition, computer resources are commonly dominated by the primitives of dynamic programming (as used in Viterbi decoding)—address calculations, reads, adds, compares, and branches (or conditional loads). This is particularly true for large vocabulary recognition. However, the use of large connectionist probability estimators can add a significant amount of computation to the recognition process. For example, for a 1000-word vocabulary we are using for recognition, a SparcStation 1+ takes roughly 10 times real time to do the dynamic programming (with no pruning of unlikely hypotheses). The neural network calculations for a large network with 300,000 connections take about 60 s on the workstation for each second of speech (for a large continuous input network). However, training via back-propagation is perhaps 5 times as long as the forward network calculation, and must be repeated over 10–20 iterations through a data set that could easily be as large as 1000 s per speaker (for the speaker-dependent Resource Management task, for instance) or 10,000 s of speech (for the speaker-independent Resource Management task). Thus, the training runs we are currently doing could take anywhere from a month to a year on a uniprocessor workstation. Planned experiments in feature selection will also require iterations over the training procedure. Since our research is largely in the area of

0743-7315/92 $3.00

training algorithms as opposed to recognition per se, a fast processor was required.

Extremely high-performance speech processing systems can be built using special-purpose VLSI designs [22, 23]. Programmable systems with somewhat lower throughput can be designed using commercial general-purpose microprocessors [4] and have the advantage of robust efficiency over a wider class of algorithms. For both approaches, custom system architectures can be used to streamline performance for a target class of algorithms. Ring architectures have been shown to be a good match to a variety of signal processing problems [5] and neural network algorithms, including back-propagation [15, 12, 24]. The Ring Array Processor design uses programmable floating-point DSP chips as the computational elements. We have connected these processors with a data distribution ring, implemented with programmable gate arrays. Some of the major hardware features are described later in this paper. For a more detailed description of the hardware design and implementation, see [2, 17].

ARCHITECTURAL CONSIDERATIONS

Artificial neural networks (ANNs) frequently do not have complete connectivity [8], even between layers of a feedforward network [16]. Nonetheless, an extremely useful subclass of these networks uses nonsparse connectivity between layers of "units," which are (for the most common case) nonlinear functions of the weighted sums of their inputs. The most common unit function uses a sigmoid nonlinearity, namely,

$$f(y) = \frac{1}{1 + e^{-y}} \tag{1}$$

with

$$y = \sum_{i=1}^{N} w_i x_i + \theta, \tag{2}$$

where the w's are connection strengths (weights), the x's are unit inputs, θ is a unit bias, y is the unit potential, and $f(y)$ is the unit output.

The computational requirements of such algorithms are well matched to the capabilities of commercial DSPs. In particular, these circuits are designed with a high memory bandwidth and efficient implementation of the "multiply-accumulate" (MAC) operation (including addressing). However, if the unit implementations and corresponding weight storage are to be divided between multiple processors, there must be an efficient means for distributing unit outputs to all of the processors. If this is not provided in the system hardware, overall operation may be extremely inefficient despite efficient arithmetic. Full connectivity between processors is impractical even for a moderate number of nodes. A reasonable design for networks in which all processors need all unit outputs is a single broadcast bus. However, this design is not appropriate for other related algorithms such as the backward phase of the back-propagation learning algorithm.

More specifically, for a forward step the weight matrix should be stored in row-major form; i.e., each processor has access to a particular row vector of the weight matrix. This corresponds to a list of connection strengths for inputs to a particular output unit. However, for a backward step the matrix should be distributed in column-major form, so that each processor has access to all connection strengths from a particular input unit. As Kung [15] has pointed out, the backward phase corresponds to a vector–matrix multiplication (as opposed to the matrix–vector multiplication of the forward case). One can use a circular pipeline or ring architecture to distribute partial sums to neighboring processors where local contributions to these sums can be added. Using this systolic mode of operation, partial sums for N units on N processors can be distributed in $O(N)$ cycles, where in contrast, a bus architecture would require $O(N^2)$ broadcasts to get all the partial sums to the processors where the complete sums can be computed. Alternatively, with a simple bus-based architecture the weight matrices could be stored twice, once in each ordering, and weight updates could be computed twice using error terms that have been distributed via broadcast. These added costs are unnecessary for the RAP because of the ring.

Table I shows the process of calculating the error terms for back-propagation on a 4-processor ring. The top section shows the initial location of the partial sums: s_{ij} refers to the ith partial sum (corresponding to the local contribution to the error term for hidden unit i) as computed in processor j. In other words, s_{ij} is all of the error term for hidden unit i which could be computed locally in processor j given the distribution of weights. In each step, each processor passes one partial sum to the processor on its right and receives a partial sum from the processor to its left (with a ring connection between the end processors). The received sum is added into one of the partial sums. By choosing the passed values correctly, all processors can be usefully employed adding in values. Thus, in the example shown, each of the four processors has a completed error sum for a hidden unit after three steps. In general, $N - 1$ steps are required to compute N such sums using N processors. Because of the ring hardware, the data movement operations are not a significant amount of the total computation, and two copies of the weights (each ordering of the weight matrix) are not necessary.

The forward calculations (as defined by Eqs. (1) and

TABLE I
Accumulation of Partial Error Sums via the Ring

P_1	P_2	P_3	P_4
	Initial partial sum location		
s_{11}	s_{12}	s_{13}	s_{14}
s_{21}	s_{22}	s_{23}	s_{24}
s_{31}	s_{32}	s_{33}	s_{34}
s_{41}	s_{42}	s_{43}	s_{44}
	Partial sum location after one ring shift		
s_{11}	s_{12}	$s_{12} + s_{13}$	s_{14}
s_{21}	s_{22}	s_{23}	$s_{23} + s_{24}$
$s_{34} + s_{31}$	s_{32}	s_{33}	s_{34}
s_{41}	$s_{41} + s_{42}$	s_{43}	s_{44}
	Partial sum location after two ring shifts		
s_{11}	s_{12}	$s_{12} + s_{13}$	$s_{12} + s_{13} + s_{14}$
$s_{23} + s_{24} + s_{21}$	s_{22}	s_{23}	$s_{23} + s_{24}$
$s_{34} + s_{31}$	$s_{34} + s_{31} + s_{32}$	s_{33}	s_{34}
s_{43}	$s_{41} + s_{42}$	$s_{41} + s_{42} + s_{43}$	s_{44}
	Partial sum location after three ring shifts		
$s_{12} + s_{13} + s_{14} + s_{11}$	s_{12}	$s_{12} + s_{13}$	$s_{12} + s_{13} + s_{14}$
$s_{23} + s_{24} + s_{21}$	$s_{23} + s_{24} + s_{21} + s_{22}$	s_{23}	$s_{23} + s_{24}$
$s_{34} + s_{31}$	$s_{34} + s_{31} + s_{32}$	$s_{34} + s_{31} + s_{32} + s_{33}$	s_{34}
s_{41}	$s_{41} + s_{42}$	$s_{41} + s_{42} + s_{43}$	$s_{41} + s_{42} + s_{43} + s_{44}$

(2)) are speeded up by employing "read-shift" hardware to distribute layer outputs with minimal processor intervention. In this scheme, the processor signals the ring hardware to pass the data on to the next ring element by the act of reading the data from the ring. Thus, to "broadcast" data from all processors to all processors, each DSP writes to the ring once and reads from it $N - 1$ times. Including overhead, the cost is

$$\text{No. cycles} = k \times (((N - 1) \times R) + W + S) + C, \quad (3)$$

where N is the number of processors, R is the number of cycles per read, W is cycles per write, S is cycles for switching between read and write modes, and C is constant overhead for a loop which iterates k times. For the broadcast of 64 unit outputs (a typical number for our application) and for the processor we have chosen (the Texas Instruments TMS320C30), this expression yields (for 16 nodes, the size of a typical system)

$$\text{No. cycles} = 4 \times (((15) \times 1) + 2 + 2) + 8 = 84 \quad (4)$$

or 1.3 cycles per unit broadcast. The constant loop overhead can be minimized with in-line coding, where necessary. The major irreducible overhead in this total is due to the effects of the internal pipeline on the DSP chip, which causes delays for external writes and for mode switching between external reads and writes.

For each board, the peak transfer rate between 4 nodes is 64 million words/s (256 Mbytes/s). This is a reasonable balance to the 64 million MAC/s (128 MFLOPS) peak performance of the computational elements. In general, units of a layer (actually, the activation calculations for the units) are split between the processors, and output activations are then distributed from all processors to all processors in the ring pseudo-broadcast described above. As long as the network is well expressed as a series of matrix operations (as in the feedforward layered case), partitioning is done "automatically" when the user calls matrix routines which have been written for the multiprocessor hardware.

HARDWARE

The RAP was intended as a working prototype and has been heavily used over the last year in our speech research. The goal was to realize a programmable system quickly with a small engineering team, but still achieve a speedup of 100× over a RISC-based workstation. As the system architecture evolved, this concern was translated into several concrete decisions:

(1) Only standard, commercially available components were used.

(2) The computational power of the system came from using several high-performance floating-point processors, not from complex interconnection schemes or sophisticated peripheral circuits. To that end, the processing nodes were kept simple to allow space for four nodes per circuit board.

(3) The memory system used only a single bank of

dynamic RAM (DRAM) and static RAM (SRAM) at each processing node. This decision permitted greatly simplified memory control logic and provided the minimum electrical loading of the processor data lines.

(4) The backplane bus chosen was a standard VME bus using only the 32 address and data lines on the basic bus interface. The RAP-specific clock signals and data distribution ring used separate connectors and cables at the front panel to avoid compromising compatibility with other VME bus products.

(5) As much of the logic as possible was implemented in Programmable Gate Arrays (PGA).[1] This decision reduced parts count, simplified board layout, and allowed flexibility for later design enhancements. PGAs were used for two functions at each node: the interprocessor communication ring and the memory controller. One additional PGA was used in the VME bus interface.

(6) The RAP is all digital and does not directly support analog I/O. By using only digital circuits, we further simplified board layout and delayed choices of the particular A/D or D/A converters used. Analog I/O is now being added without redesigning the RAP by using connectors provided for access to the DSP serial ports. These high-speed ports can support data conversion rates as high as 1 Mbyte/s.

Figure 1 shows the major elements of a RAP node. Each node consists of a DSP chip, the local memory associated with that chip, and the ring bus interface, incorporating a simple two-register pipeline and handshake logic for ring communication control. Four interconnected nodes are contained on a single circuit board, with the internode buses also brought to connectors at the board front edge. At the next higher level, one or more boards are plugged into a common VME backplane with all ring buses connected using flat ribbon cable. A Sun CPU board host is plugged into the same VME bus, providing the primary control of the RAP, as well as disk and Ethernet services. The user interface is either through the console of the Sun host or from other workstations over the network via a daemon running on the Sun.

A critical element of the design was the PGA implementation of the ring bus interface and handshake mechanism. The bus interface PGA, with 144-user-programmable I/O connections, hides all of the wiring complexity associated with the ring data paths, yet is fast enough to implement short protocols in a single RAP cycle. However, because of the availability of 640 reconfigurable logic blocks within the PGA, modifications to the data transfer protocol can be added at a later time. One anticipated modification is to use spare "tag" signal lines with unused PGA logic blocks to implement more complex memory addressing protocols.

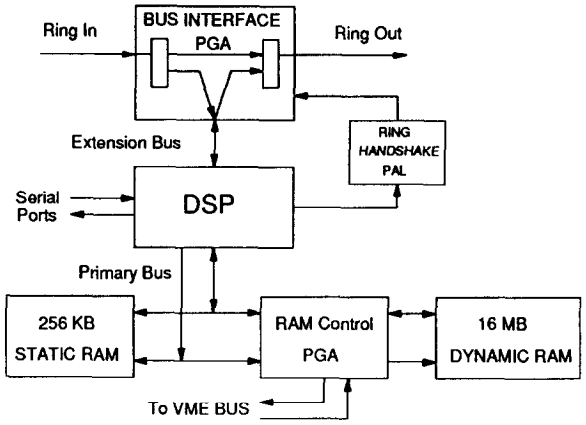

FIG. 1. Block diagram for a single RAP node.

Each ring interface includes an input and an output data register, providing the minimum buffering needed for sustained performance. Handshaking for the ring uses two signals between adjacent pairs of nodes under the control of a 7.5-ns PAL[2] at each node. The PAL is responsible for generating the time-critical READY signal required by the DSP, based on the contents of four data registers (two in the local node and one each in the neighboring nodes). This handshake-READY interlock mechanism provides a hardware-level block synchronization that is well suited to our SIMD style of programming for the RAP.

The synchronous operation of the ring data transfers requires a low-skew master clock for handshake control at each of the nodes. Although the logic and registers all operate at a crystal-controlled frequency of 16 MHz, both edges of the clock are used in the design. We distribute a square wave clock at ECL logic levels over a short private cable along the board front edge. Separate ECL-to-TTL receivers drive this clock to the reference input of a phase-locked loop (PLL) at each node. Each PLL in turn controls the input clock to a node DSP, closing the loop and eliminating skew due to DSP variations. Measurements on a 40-node RAP system show all internode skews to be less than 4 ns.

The RAM controller is also implemented using a PGA to conserve board space and enhance functionality. This device provides the timing and address multiplexing functions required by the DRAMs, along with the data paths and logic required for node memory access from the VME bus. Provision was also made for the RAM controller to implement memory error control, using parity and ECC if required.

The interface to the VME bus uses a pair of special-purpose control chips, some address and data buffers,

[1] PGA is a registered trademark of Xilinx Incorporated.

[2] PAL is a registered trademark of Advanced Micro Devices.

and a PAL for address decoding. A PGA is used in the VME interface to implement control and status registers for the individual RAP nodes, along with board initialization functions. The RAP allows full 32-bit reads and writes directly to node RAM, with hardware-level control of the DSP to prevent contention. From the VME side, all node memory is uniquely mapped ("flat" addressing), allowing simple access from the host CPU. The address decoding PAL locates the board within the VME address space, with a current limit of 16 boards on one host.

The design used a small number of component types to minimize complexity. There is very little visible "glue" logic, despite the overall pin count (over 10,000 holes per board), because of the extensive use of PGAs. Thus, out of 59 parts within a node, all except for two PALs, three address buffers and a clock oscillator are memory, gate arrays, or the DSP itself. This simplicity has made the implementation of a powerful system tractable for a small group with limited resources.

BENCHMARK RESULTS

Three six-layer printed circuit boards were fabricated in late 1989, and low-level software and firmware were written to bring the system to a usable state by mid-1990. The boards were programmed to implement common matrix–vector library routines, and to do the forward and backward phases of back-propagation. More recently (early 1991), 10 boards were fabricated and tested in a single card cage. Results on this larger system with a simple benchmark network with one hidden layer and all layers of the same size are shown in Table II. Matching layer sizes are not required by either the hardware or the neural net software, but this choice permits a simplified analysis of the machine performance as a function of the number of units in a layer.

The first row of Table II shows the performance for a subtask that is close to optimal for the TI DSP. For a large enough dimension, this routine exceeds 80% efficiency (with respect to 64 MMACS/board for a 16-MHz clock), and 10 RAP boards are roughly 600 times the speed of a Sun SparcStation 2 running the same benchmark. For the larger networks, the forward propagation performance becomes almost identical to the matrix–vector multiply (which is $O(N^2)$), since the sigmoid calculation (which is $O(N)$) becomes inconsequential in comparison to the multiply-accumulates. Finally, when learning is performed on each cycle (for a network with one hidden layer), the weight update steps dominate the computation. This is commonly the case with the back-propagation algorithm, and similar ratios have been reported for other multiprocessor implementations [12, 9, 28]. Each update and delta calculation requires at least as many arithmetic operations as the forward step, so that a factor of 3–5 decrease in throughput should be expected for the network calculation when learning is included. Another limitation is the DSP, which is optimized for dot-product calculations rather than the read–add–write of the weight update step. For the complete forward and backward cycles, the 57–102 million connection updates per second (MCUPS) shown in the table corresponds to a speedup over the SparcStation 2 of about 100–200. For an average of five floating-point arithmetic operations per connection during learning, the last line of Table II corresponds to 285–510 MFLOPS, or roughly one-fourth to one-half of the peak arithmetic capability of the machine. For forward propagation, with an average of two floating-point operations per connection, 90% of the peak arithmetic capability is obtained for the larger problem.

Figures 2 and 3 further illustrate the benchmark performance of 1-, 5-, and 10-board RAP systems. As the prob-

TABLE II
Uniform Layer Benchmarks, 10-Board RAP

Operation	Matrix size	
	128 × 128	640 × 640
Matrix × vector	266.4 MCPS	573.4 MCPS
Forward propagation	211.1 MCPS	558.3 MCPS
Forward plus learning	57.3 MCUPS	102.1 MCUPS

Note. MCPS, Millions of connections per second; MCUPS, millions of connection updates per second.

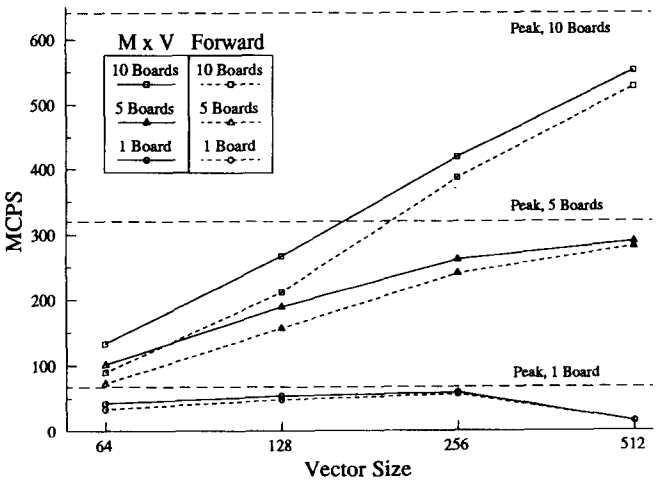

FIG. 2. RAP performance for forward propagation from one layer to the next, with and without a nonlinear (sigmoidal) transformation. For the linear case (the higher in each pair of curves), this is equivalent to matrix–vector multiplication. The horizontal lines show peak performance for each RAP size. The measure is millions of connections per second (MCPS). The performance reduction for the last data point is due to running out of SRAM to hold the weights on a single-board system, so that DRAM must be used.

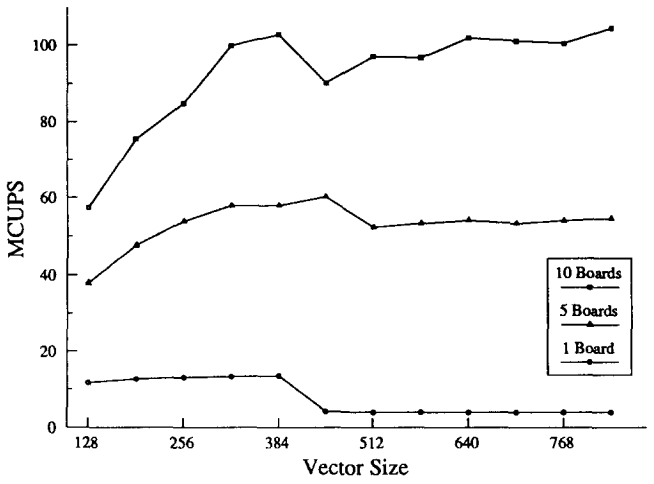

FIG. 3. RAP performance for full learning cycle in a network with one hidden layer in millions of connection updates per second (MCUPS). The step down in performance for moderate vector sizes is due to running out of on-chip RAM to hold network variables, so that off-chip SRAM must be used.

lems increase in size, performance approaches peak computation rates and demonstrates a closer approximation to a linear speedup. Efficiency of parallelism is considered further in the next section.

Table III shows the 10-board RAP performance for two feedforward networks with nonuniform layer sizes. The first row corresponds to a common autoassociative network, and the second to an architecture that we have used in our speech recognition training. For these examples, performance is in the range 341–429 million connections per second (MCPS) for forward propagation alone, and 45–83 MCUPS for the complete learning cycle for one pattern.

SYSTEM PERFORMANCE FACTORS

Extrapolation of performance measurements to a larger system is obviously problem-dependent, but what we have observed with 1–10 boards is a speedup that is close to linear for layers of size $\geq 4\times$ the number of processors. For the 32-bit host address space, the largest possible system would consist of 16 RAP boards (64 nodes), would have over 1 GB of memory, and would have a peak performance of 2 GFLOPS. Our simulations project a forward propagation performance in excess of 800 MCPS for such a hypothetical machine. Larger RAP systems could be built for a host with a larger address space. The size of the largest useful RAP is ultimately limited by the effect of parts of the application which cannot be partitioned between processors. For back-propagation learning in a layered feedforward network with one hidden layer, and an equal number of units in each layer, let

α_Q = number of cycles per connection (partitioned)

α_{L1} = number of cycles per unit (partitioned)

α_{L2} = number of cycles per unit (not partitioned)

C = constant overhead

N = number of units

P = number of processors

W = number of words of memory available for connection weights;

then the total time in cycles for the algorithm would be

$$\alpha_Q \frac{N^2}{P} + \alpha_{L1} \frac{N}{P} + \alpha_{L2} N + C. \tag{5}$$

For large values of N (where N/P is held constant), we can ignore the second and fourth terms, so the machine would be 50% efficient or better when

$$\alpha_Q \frac{N^2}{P} \geq \alpha_{L2} N \tag{6}$$

or

$$P \leq N \frac{\alpha_Q}{\alpha_{L2}} \tag{7}$$

squaring and dividing by P,

$$P \leq \frac{N^2}{P} \frac{\alpha_Q^2}{\alpha_{L2}^2}. \tag{8}$$

For the case of the maximum size weight matrix to fit in the static memory, this becomes

$$P \leq W \frac{\alpha_Q^2}{\alpha_{L2}^2}. \tag{9}$$

From our test runs, we have estimated α_Q to be 9 for the back-propagation calculation (7 when the weights are in internal memory) and α_{L2} to be 21, where the latter come

TABLE III
Nonuniform Layer Benchmarks, 10-Board RAP

	Autoassociation 512 → 256 → 512	MLP for HMM 234 → 1024 → 61
Forward propagation	429 MCPS	341 MCPS
Forward plus learning	83 MCUPS	45 MCUPS

from the delta calculation for the hidden layer, and from communication overhead. Therefore, an upper bound on processors for this problem appears to be roughly

$$P \leq W/4. \tag{10}$$

Since our design includes 64K words of fast static RAM per node, the RAP would scale up to about 16,000 processors (for sufficiently large back-propagation problems). At this point the code that was linear in the number of units (such as communication costs, and the outer loop of the delta calculation for hidden units) would dominate the computation. Obviously, many practical problems such as synchronization would be much more difficult for such a massively parallel version. Furthermore, the training time scales badly with large back-propagation nets, so that large machines should actually be analyzed for a more likely topology for a large network, for instance, with sparse connectivity.

Substituting the observed values for α_Q and α_{L2} into (8), we get roughly

$$\frac{N}{P} \geq 2. \tag{11}$$

This latter limit is of more immediate and practical concern. The RAP, used as a back-propagation machine, is reasonably efficient for networks with at least 2 units represented per processor. For small values of P (e.g., 4), a more detailed analysis (considering all four terms) suggests a preferred ratio of $N/P \geq 4$.

Estimating the constants from our measurements with the RAP, the machine cycles required to process one pattern (including learning on a P-processor RAP) is

$$9\frac{N^2}{P} + 92\frac{N}{P} + 21N + 2336. \tag{12}$$

This is a good match to our measurements for values of N in the range 128–512, although the first coefficient goes to 7 for cases in which the weights all fit in internal memory. For large values of N, the limited storage internal to the DSP chip reduces performance further. For most examples in our experience, the contributions to the linear terms from communication are negligible. As mentioned earlier, the largest inefficiency in the system is the time required for weight updates, which typically dominates the α_Q term.

Using Eq. (12), we can infer the efficiency of parallelism for our target algorithm on a RAP of various sizes, as shown in Figs. 4 and 5. This efficiency is the fraction of linear speedup achieved with P processors, for back-propagation on a layered network with N units in an input, single hidden, and output layer. This equation was derived from a combination of known routine lengths and clock measurements with a single board, and was found to be an extremely close match to later measurements with boards. For the problem sizes of interest to us in our speech work, for which the number of units per layer is typically 64 to 1024, systems in the range of 4–40 processors (1–10 boards) give a respectable efficiency (45–90%).

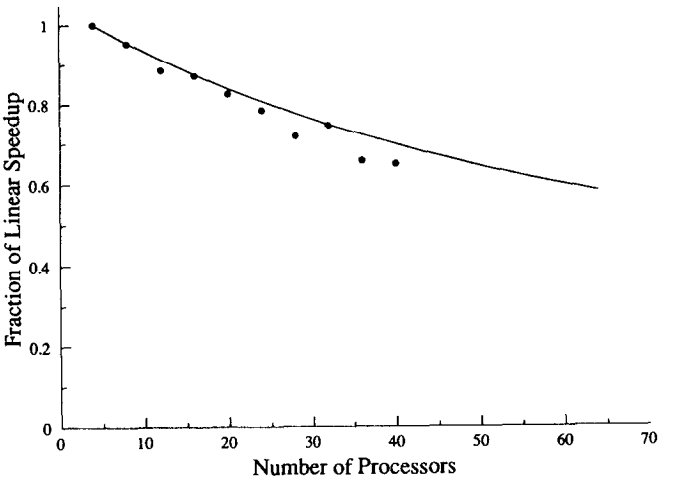

FIG. 4. Estimated fraction of linear speedup for uniform-sized layers, same network as that in Fig. 3 with 256 hidden units. The data points reflect actual measurements, while the curve is derived from inspection of the code and observation of run times. The measurements deviate from the ideal curve for problem sizes that are not integer multiples of the number of processors. For these cases, the layers are essentially padded with zero values.

For much larger systems, assuming a host with a larger address space, projected performance on back-propagation is shown in Fig. 5. Certainly problems with less than 1000 units per layer would underutilize such a mammoth

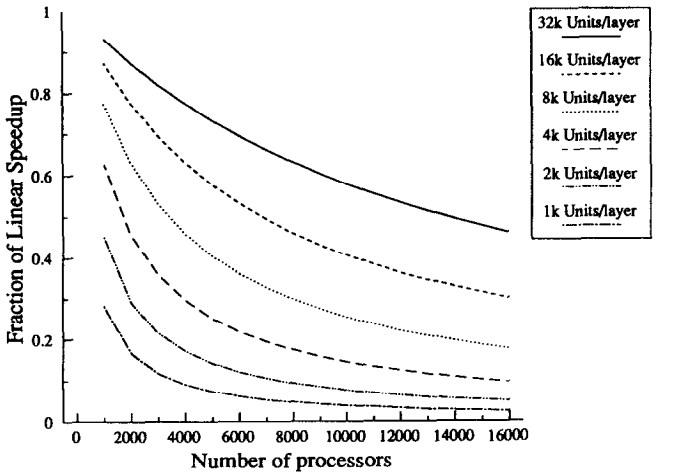

FIG. 5. Estimated fraction of linear speedup for uniform-sized layers, much larger idealized RAP.

machine. For each possible machine size, roughly 50% performance can be achieved on a problem with layer size of $2P$. It is actually unlikely that such a large uniform network would really be practical, simply for reasons of training. The ring topology may not be a good match to a sparsely connected net, but the trade-offs are still unknown since our current software is unsuitable for such a problem. We intend to investigate this important point in future work.

SOFTWARE DESIGN

The goals of the RAP software design were:

- Efficiency for computational tasks of interest (back-propagation and speech recognition algorithms).
- Ease of use by speech scientists who are often not familiar with object-oriented languages.
- Similarity to the standard UNIX environment (allow program debugging under UNIX).
- A short path to operational system software.

The RAP now runs in VME-based systems including Sun workstations under UNIX (e.g., a Sun 4/330) and as an attached processor for a single-board computer running the VxWorks real-time operating system. The latter software was especially useful in early stages of debugging the RAP, while the former provides a familiar user environment and the usual panoply of device drivers for peripherals such as SCSI disks. An extensive set of software tools has been designed and implemented. Primary emphasis was placed on improving the efficiency of layered artificial neural network algorithms. This was done by providing a library of matrix-oriented assembly language routines, some of which use node-custom compilation (described later in this section). An object-oriented RAP interface in C++ that allows programmers to incorporate the RAP as a computational server into their own UNIX applications is provided. For those not wishing to program in C++, a command interpreter that provides interactive and shell-script style RAP manipulation has been built.

Although the RAP hardware is capable of MIMD operation (Multiple Instruction streams controlling Multiple Data streams), the software and communications ring were designed to expedite SPMD (Single Programs operating on Multiple Data streams) style of programming. This is a varient of SIMD programming in which instructions are not lock-stepped. In SPMD programming the same program is loaded into all of the processors. Usually, the processors will all be performing the same operations on different parts of the data, although in some cases this may not occur simultaneously. For example, to multiply a matrix by a vector, each processor would have its own subset of the matrix rows that must be multiplied. This is equivalent to partitioning the output vector elements among the processors. If the complete output vector is needed, the ring broadcast routine is called to redistribute the part of the output vector from each processor to all the other processors.

Since there is no shared memory between processing nodes, all interprocessor communication is handled by the ring. The hardware does not automatically keep the processors in lock step; for example, they may become out of sync because of branches conditioned on the processor's node number. However, when the processors must communicate with each other through the ring, hardware synchronization automatically occurs (using the local hardware handshake referred to earlier). A node that attempts to read before data are ready or write when there are already data waiting will stop executing until the data can be moved.

The three primitives used to communicate between nodes are **ring_get( ), ring_put( ),** and **ring_shift( ). ring_get( )** reads a data word from the previous node, or blocks until one is ready. **ring_put( )** writes a data word to the subsequent node, or blocks until that node reads. **ring_shift( )** reads a data word from the previous node and writes that word to the subsequent node, or blocks until the previous node has written and the subsequent node has read. These routines provide all the RAP's synchronization primitives.

Any UNIX application that manipulates the RAP is called a **rapClient.** A **rapClient** connects to a **rapServer** (or RAP host) which resides on the machine hosting a RAP system (e.g., if a RAP board is in a Sun VME bus backplane, a **rapServer** daemon process will wait for connections from **rapClients** and acknowledge interrupts from the RAP board). Any user-written UNIX application may become a **rapClient** by inheriting from the right C++ class.

A **rapClient** communicates to a **rapServer** using the RAP Client–Host Protocol, which performs both presentation and application level functions. Built on top of TCP/IP, the protocol provides:

- Data type safety: type check the sequence of data messages sent to the **rapServer.** The **rapServer** will discard a RAP message if it does not strictly follow the format.
- Floating-point number conversion: floating-point numbers on the **rapClient** end of the connection must be in IEEE standard format, but on the server end must be in TMS320C30 format. A **rapClient** is assured that floating-point numbers sent to a **rapServer** in IEEE format will be converted appropriately for each direction.
- Virtual RAP node numbers seen by the **rapClient:** the **rapClient** will only see node numbers 0 through *numberOfNodes* − *1* regardless of the physical node numbers.

• A small amount of physical security: the current user's job may not be disturbed by any other user.

User applications running on a Sun that are written in C++ may inherit from the C++ **rapClient** class and may then send protocol messages to the RAP by calling **rapClient** member functions. An application can thus use the RAP as a computational server for an interactive graphics application (although no form of real-time is guaranteed) or for a UNIX filter.

A **rapClient** references nodes using virtual node numbers. A virtual node number is translated by the **rapServer** to a hardware (board ID, node ID) tuple. With virtual node numbers, a new configuration of hardware board ID and node ID RAP boards may be installed in any order, and the user will only see consecutive node numbers starting at zero.

RAPMC is a **rapClient** that is used for interactive controlling and debugging of a RAP program. By sending sets of RAP Client–Host Protocol requests for each command entered at the terminal, RAPMC allows interactive control of the RAP directly from the RAPMC command line. It also gives users the ability to change the destination of RAP standard and error output. Users may start up multiple RAPMCs and have different RAP nodes send output to each RAPMC.

RAPMC supports interactive commands to:

- Load and run RAP programs.
- Reset RAP nodes.
- Display the RAP user queue.
- Disassemble TMS320C30 instructions.
- Examine or modify RAP memory.
- Redirect RAP node output to a file or UNIX process.
- Execute Command Scripts.
- Send ASCII text to a RAP node's standard input (stdin).
- Wait for RAP jobs to complete.

See [14] for a complete user's reference on RAPMC.

Many standard C routines are provided in the UNIX style C programming environment. Some examples are:

- file I/O such as open, fopen, read, fread, etc.
- memory allocation routines
- argc and argv for command line arguments
- global variables stdin, stdout, and stderr.

Additionally, a library of assembly language routines provides many common matrix and vector routines, such as matrix times vector, matrix times matrix, and Euclidean distance.

Two global node constants are provided: N_NODES (the number of nodes in the RAP system, determined automatically at power-up) and NODE_ID for each node (also determined at power-up). These constants must be used by the ring distribution routines so that each node knows which part of a replicated array to read from the ring. This could be done most simply by computing the proper array location at each iteration of an inner loop consisting of one **ring_put( )** following by N_NODES-2 **ring_shift( )** instructions. However, this requires a large amount of extra integer computation and would slow down the **ring_distribute( )** routine appreciably. Another time-intensive solution would be to use indirect references to code stubs; this is problematic because of pipeline delays that are incurred in the DSP chip for this kind of repeated use of address registers. Still another solution would be to store different versions of the ring routines for all plausible values for N_NODES and NODE_ID. This would be time-efficient, but extremely wasteful of space.

An alternative solution is to compile the correct version of the routine once at run-time. Part of the bootstrap code calls a ring initialization routine, **ring_init( ).** Inside **ring_init( ),malloc( )** is called to allocate a buffer for the customized ring code. Then, templates for the instructions in the loop are copied into the buffer. The number and order of these instructions depend on NODE_ID and N_NODES. The only instruction template that must be modified before being copied is the loop size field of the TMS320C20 repeat block instruction.

Because the code is generated into an allocated data buffer of the correct size, there are none of the dangers involved in patching code generated by the assembler. This run-time "custom-compilation" approach has the efficiency of a space-intensive code repetition solution, but uses the same space as the simpler more time-intensive approaches.

RAP APPLICATIONS

The prototype system has now been used for a series of studies in feature extraction for continuous speech recognition [20, 21]. We have experimented with a number of connectionist architectures, but one of the most successful has been a simple but large network that was referred to above as the 234 → 1024 → 61 architecture. This net consisted of the following components:

(1) Input layer—nine groups of 26 inputs, one for the current 10-ms frame of speech and one for each of four frames into the past and future (i.e., a nine-frame window of temporal context). The inputs for each frame were 13 coefficients from a Perceptual Linear Prediction (PLP) analysis [11] and 13 coefficients from an estimate of the instantaneous temporal slope of these coefficients.

(2) Hidden layer—1024 units that receive connections from all input units. Experiments showed that significant performance could be seen for increases in hidden layer size for up to this number.

(3) Output layer—61 units, corresponding to 61 pho-

netic classes, receiving connections from all hidden units.

While clever shortcuts could have reduced the size of this net, it was an enormous convenience not to have to be concerned with such matters while doing the feature explorations. Additionally, we got our speedup (overnight on the RAP rather than 1–2 months on the Sun) without any special programming; for most of our experiments, we used a standard layered network program that was designed at ICSI early in 1990.

As described above, the primary target application for the RAP was back-propagation training of layered neural networks. However, we did not build special-purpose integrated circuits, which could have had a considerable speed advantage, because we wanted a programmable machine. While our current uses for the RAP continue to be dominated by back-propagation, we are able to modify the network algorithm daily for our experiments. Furthermore, we have experimented with using the RAP for computations such as the generation of Mandelbrot and Julia sets, computation of radial basis functions, and optimization of multidimensional Gaussian mixtures by a discriminative criterion, and for dynamic programming. We also have used the RAP for calculation of dynamic features (first, second, and third temporal derivatives) to feed the layered network. While the topology has been optimized for the block matrix operations required in back-propagation, many algorithms can benefit from the fast computation and communication provided by the RAP.

In the case of dynamic programming, for instance, we currently treat a board as four independent processors that perform recognition on different sentences, thus speeding up a batch run by four. For real-time operation, the reference lexicon would be split up between processors, so that processors only need to communicate once for each speech frame. Thus, the RAP can be used as a SIMD or SPMD machine (for our matrix operations, as in back-propagation), as a farm of separate SISD machines, requiring essentially no intercommunication (as in our current use for off-line dynamic programming), or as a MIMD machine with simple and infrequent communication (as in the dynamic programming case for a distributed lexicon). Software is being developed to support all of these modes [3, 14, 17].

Further C++ software is being developed for the RAP using the C++ preprocessor from AT&T. In this code, vectors or matrices are object classes that inherit from a more general class of distributed data object. The distribution of data is encapsulated so that SIMD or SPMD programmers can use the machine as if it were one large array processor with a large shared memory. Additionally, we are building a new back-propagation training program called CLONES (Connectionist Layered Object-oriented NEtwork Simulator) that runs on the RAP and SUN workstation [13].

RELATED WORK

The simple communication ring topology used in the RAP is common to several other proposed and realized machines. Similar examples are the NeuroTurbo from Nagoya University [12] and the WARP [1, 24]. The RAP differs from these most significantly in its use of PGAs for communications hardware. These arrays permit easy modification of the low-level register transfer logic to provide flexibility without sacrificing speed. In the iWARP (a commercial machine inspired by the WARP) a versatile communications processor performs data transfers between computing nodes with a latency of 100–150 ns per word. The programmability of this communications processor favors the iWARP for systems using high-level complex protocols. The custom VLSI circuits used in the iWARP provide 60% of the computational capability of the TMS320C30 used in the RAP, with a significantly more sophisticated communications capability. The RAP, on the other hand, transfers words between DSP nodes within a single 62.5-ns cycle using a very simple PGA design. In addition, the ability to customize this array for different low-level communication protocols without sacrificing performance is an advantage for the RAP. Back-propagation has been mapped to the WARP in a manner somewhat different than that reported here [24]. Users of that machine chose to pass partial sums for the forward pass rather than the backward pass of our case; that is, they apparently stored the weights corresponding to the outputs of each unit, rather than the input. Another approach that they tried was to use each processor for a different copy of the complete network. Each copy operated on different segments of the data; they ran multiple forward passes without updating the weights, and thus could read the weights in from a larger central memory. The resulting delay between forward passes and updates is probably acceptable for most applications, but it was not necessary on the RAP because of the large amount of fast local memory.

In contrast to both the RAP and the WARP, the NeuroTurbo uses dual-port memories between pairs of processors to implement the communication ring. The dual-port memory approach provides no hardware communication interlock but presents a simple model for user-designed software management of the ring. The designers apparently used a similar approach to the technique illustrated in Table I, but passed partial sums around the ring for the forward rather than the backward pass of the algorithm.

Two prominent examples of digital neural network integrated circuits that have been announced are the systolic Neuroemulator of Siemens [25] and the "X1" chip

from Inova and Adaptive Solutions [10]. Although these will be comparatively fixed-function machines, they are expected to have significantly higher performance than the currently available alternatives, at least for the more common layered network algorithms.

SUMMARY

Ring architectures have been shown to be a good match to a variety of signal processing and connectionist algorithms. We have built a Ring Array Processor using commercial DSP chips for computation and Programmable Gate Arrays for communication. Measured performance for a 10-board system on target calculations is two to three orders of magnitude higher than we have achieved on a general-purpose workstation. A programming environment has been provided so that a programmer may use familiar UNIX system calls for I/O and memory allocation. This tool is greatly aiding our connectionist research, particularly for the training of speech recognition systems, allowing exploration of problems previously considered computationally impractical.

ACKNOWLEDGMENTS

Critical design review was provided by Berkeley scientists and engineers too numerous to name, but certainly Jerry Feldman, Steve Omohundro, Jan Rabaey, and Joel Libove should be mentioned. Joel Libove also provided a more detailed hardware design review at critical stages. Herve Bourlard (formerly of Philips, now with L&H Speechproducts) provided the theoretical foundations for the speech application, and continues to collaborate with us on this work. Hynek Hermansky of US West has collaborated with us on the evaluation of perceptually relevant feature extraction algorithms for speech recognition. Chuck Wooters was our first nondeveloper RAP user, and worked to apply the RAP routines to our speech problems. Components were contributed by Toshiba America, Cypress Semiconductor, and Xilinx Inc., and Texas Instruments provided free emulators to debug the DSPs in-circuit. Finally, support from the International Computer Science Institute for this work is gratefully acknowledged.

REFERENCES

1. Annaratone, M., Arnould, A., Kung, H. T., and Menzilcioglu, O. Using warp as a supercomputer in signal processing. *ICASSP '86 Proceedings*, Tokyo, pp. 2895–2898.
2. Beck, J. The ring array processor (RAP): Hardware. International Computer Science Institute TR-90-048, 1990.
3. Bilmes, J., and Kohn, P. The ring array processor (RAP): Software architecture. International Computer Science Institute TR-90-050, 1990.
4. Bisiani, R., Anantharaman, T., and Butcher, L. BEAM: An accelerator for speech recognition. *Proc. IEEE Intl. Conf. on Acoustics, Speech, & Signal Processing*, Glasgow, Scotland, pp. 782–784, 1989.
5. Bourlard, H., and Morgan, N. Merging multilayer perceptions and hidden Markov models: Some experiments in continuous speech recognition. International Computer Science Institute TR-89-033, 1989.
6. Bourlard, H., Morgan, N., and Wellekens, C. J. Statistical inference in multilayer perceptrons and hidden Markov models with applications in continuous speech recognition. *Neuro Computing, Algorithms, Architectures and Applications*, NATO ASI Series, 1990, Vol. F68, pp. 217–226.
7. Bourlard, H., and Wellekens, C. J. Links between Markov models and multilayer perceptrons. In *Advances in Neural Information Processing Systems*. Morgan Kaufmann, San Mateo, 1989, Vol. 1, pp. 502–510.
8. Feldman, J. A., Fanty, M. A., Goddard, N., and Lynne, K. Computing with structured connectionist networks. *Comm. ACM*, 1988.
9. Garth, S. A chipset for high speed simulation of neural network systems. *First International Conference on Neural Networks*, San Diego, June 1987, pp. III-443–452.
10. Hammerstrom, D. A VLSI architecture for high-performance, low-cost, on-chip learning. *Proc. IJCNN*, San Diego, June 1990.
11. Hermansky, H. Perceptual linear predictive (PLP) analysis of speech. *J. Acoust. Soc. Amer.* **87,** 4 (Apr. 1990).
12. Iwata, A., Yoshida, Y., Matsuda, S., Sato, Y., and Suzumura, N. An artificial neural network accelerator using general purpose floating point digital signal processors. *Proc. JCNN*, 1989, pp. II-171–175.
13. Kohn, P. CLONES: A connectionist layered object-oriented network simulator. ICSI TR, in preparation.
14. Kohn, P., and Bilmes, J. The ring array processor (RAP): Software users manual. International Computer Science Institute TR-90-049, 1990.
15. Kung, S. Y., and Hwang, J. N. A unified systolic architecture for artificial neural networks. *J. Parallel Distrib. Comput.* (Apr. 1989).
16. Le Cun, Y., Denker, J., Solla, S., Howard, R., and Jackel, L. Optimal brain damage. In David Touretzky (Ed.). *Advances in Neural Information Processing Systems*. Morgan Kaufmann, San Mateo, 1990, Vol. 2.
17. Morgan, N., Beck, J., Kohn, P., Bilmes, J., Allman, E., and Beer, J. The RAP: A ring array processor for layered network calculations. *Proc. Intl. Conf. on Application Specific Array Processors*. IEEE Computer Society Press, Princeton, NJ, 1990 pp. 296–308.
18. Morgan, N., and Bourlard, H. Generalization and parameters estimation in feedforward nets: Some experiments. International Computer Science Institute TR-89-017.
19. Morgan, N., and Bourlard, H. Continuous speech recognition using multilayer perceptrons with hidden Markov models. *Proc. IEEE Intl. Conf. on Acoustics, Speech, & Signal Processing*, Albuquerque, NM, 1990, pp. 413–416.
20. Morgan, N., Hermansky, H., Wooters, C., Kohn, P., and Bourlard, H. Continuous speech recognition using PLP analysis with multilayer perceptrons. *IEEE Intl. Conf. on Acoustics, Speech, & Signal Processing*. Toronto, Canada, 1991, pp. 49–52.
21. Morgan, N., Wooters, C., Bourlard, H., and Cohen, M. Continuous speech recognition on the resource management database using connectionist probability estimation. ICSI TR-090-044; also to be published in *Proc. ICSLP-90*, Kobe, Japan.
22. Murveit, H., and Brodersen, R. W. An integrated-circuit-based speech recognition system. *IEEE Trans. Acoust. Speech Signal Process.* **ASSP-34,** 6 (Dec. 1987).
23. Murveit, H., Mankoski, J., Rabaey, J., Brodersen, R. W., Stoelzle, T., Chen, D., Narayanaswamy, S., Yu, R., Schrupp, P., Schwartz, R., and Santos, A. A large-vocabulary real-time continuous-speech recognition system. *Proc. IEEE Intl. Conf. on Acoustics, Speech, & Signal Processing*, Glasgow, Scotland, 1989, pp. 789–792.

24. Pomerleau, D., Gusciora, G., Touretzky, D., and Kung, H. T. Neural network simulation at warp speed: How we got 17 million connections per second. *IEEE International Conference on Neural Networks*, San Diego, CA, July 1988.

25. Ramacher, U., and Raab, W. Fine-grain system architectures for systolic emulation of neural algorithms. *Proc. Intl. Conf. on Application Specific Array Processors*. IEEE Computer Society Press, Princeton, NJ, 1990, pp. 554–566.

26. Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation. In Rumelhart, D. E., and McClelland, J. L. (Eds.). *Parallel Distributed Processing. Exploration of the Microstructure of Cognition*, Vol. 1, *Foundations*. MIT Press, Cambridge, MA, 1986.

27. Werbos, P. J., Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. thesis, Department of Applied Mathematics, Harvard University, 1974.

28. Zhang, X. An efficient implementation of the back-propagation algorithm on the connection machine CM-2. In David Touretzky (Ed.). *Advances in Neural Information Processing Systems*. Morgan Kaufmann, San Mateo, 1990, Vol. 2.

Received August 19, 1991; revised September 8, 1991; accepted September 12, 1991