arallel neural computer," *Connectionism in a Broad Perspective: Selected Papers from the wedish Conference on Connectionism - 1992,* Niklasson and Bodén Eds. Ellis Horwood, p. 213-226, 1994. ISBN 0-13-176751-8

TOWARDS MODULAR, MASSIVELY PARALLEL NEURAL COMPUTERS

Bertil Svensson
Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden and Centre for Computer Science, Halmstad University, Halmstad, Sweden
email: svensson@ce.chalmers.se

Tomas Nordström
Division of Computer Science and Engineering, Luleå University of Technology, Luleå, Sweden
email: tono@sm.luth.se

Kenneth Nilsson and Per-Arne Wiberg
Centre for Computer Science, Halmstad University, Halmstad, Sweden
email: Kenneth.Nilsson@ite.hh.se, Per-Arne.Wiberg@cdv.hh.se

ABSTRACT

A new system-architecture, incorporating highly parallel, communicating processing modules, is presented as a candidate platform for future high-performance, real-time control systems. These are needed in the realization of action-oriented systems which interact with their environments by means of sophisticated sensors and actuators, often with a high degree of parallelism, and are able to learn and adapt to different circumstances and environments. The use of artificial neural network algorithms and trainability require new system development strategies and tools. A Continuous Development paradigm is introduced, and an implementation of this, in the form of an interactive graphical tool, is outlined. The architectural concept is based on resource adequacy, both in processing and communication. Learning algorithms are cyclically executed in distributed nodes, which communicate via a shared high-speed medium. The suitability of SIMD (Single Instruction stream, Multiple Data streams) processing nodes for ANN computations is demonstrated. An implementation of the system architecture is presented, in which distributed SIMD-nodes access their data from local real-time databases, updated with data from the other nodes via a shared optical link.

Keywords: Parallel processing; learning systems; neural networks; action-oriented systems, control system design, real-time computer systems.

1 INTRODUCTION

"Action-oriented systems", as described by Arbib [Arbib, 1989], interact with their environments by means of sophisticated sensors and actuators, often with a high degree of parallelism. The ability to learn and adapt to different circumstances and environments are among the key characteristics of such systems. Development of applications based on action- oriented systems relies heavily on training, rather than progamming of the detailed behaviour.

Response time requirements and the demand to accomplish the training task point to massively parallel computer architectures. A network of homogeneous, highly parallel modules is foreseen. The modules perform perceptual tasks close to the sensors, advanced motoric control tasks close to the actuators, or complex calculations at "higher cognitive levels". The new system-architectural concept that we introduce for the implementation of this kind of highly parallel real-time systems is based on the principle of resource adequacy [Lawson, 1992b] in order to achieve predictability. This means that enough processing and communication resources are designed into the system and statically allocated to guarantee that the maximum possible work-load can always be handled.

Not only do these trainable control systems require new architectural paradigms, they also require the acceptance of new system development philosophies. The traditional application-development model, characterized by a sequence of development phases, must be replaced by an interactive model based on training.

Both the system development model and the architectural paradigm are first presented on the conceptual level and then examplified by describing implementations meeting the demands of typical advanced real-time control tasks. Specifically, this paper points to the possibilities based on multiple SIMD (Single Instruction stream, Multiple Data streams) arrays on which static allocation of processing tasks is made and on the power and appeal of graphical application-development tools.

We have shown, by own implementations and detailed studies, as well as by reviewing the implementations of others, that typical neural network algorithms used today map efficiently onto SIMD architectures [Nordström and Svensson, 1992]. Based on this, and the discussion above, a hypothetical architecture for Artificial Neural Systems (ANSs) would look like the one shown in Figure 1.

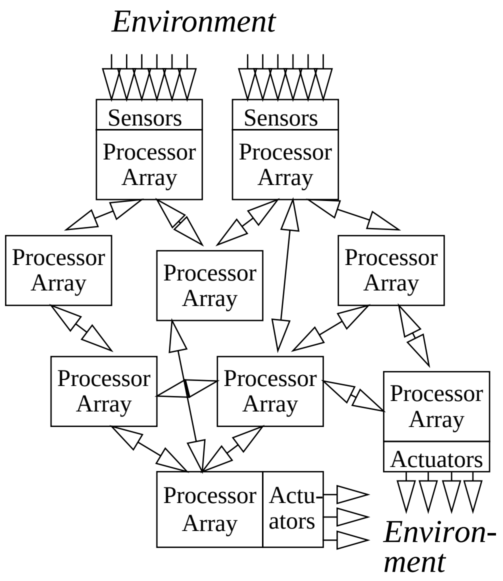

Figure 1. A multi-module architecture for an action-oriented system

Different modules (SIMD arrays) typically execute different Artificial Neural Network (ANN) models, or different instances of the same model. Full connectivity may be used within the modules, while the communication between modules is expected to be less intensive (although we will also devise solutions that satisfy the potential demand for tighter connections between pairs of modules).

The work is part of REMAP[3], the Real-Time, Embedded, Modular, Action-oriented, Parallel Processor Project, partly funded by STU/NUTEK, the Swedish National Board for Technical and Industrial Development, under contracts No. 9001583 and 9001585.

2 LEARNING ALGORITHMS AND MODULE ARCHITECTURE

Studies of the brain indicate that adaptation takes place in basically two ways: by changing the structure and by changing the synapses (connection strengths in the structure). The first one has the nature of long-term adaptation and often takes place in the first part of an animal's life. The second one, the changes of connection weights (the synapses), is a more continuous process and happens throughout the animal's entire lifetime.

Modeled after this, the design of an action-oriented system should first be concerned with the process of selecting and connecting (possibly adapting) ANN structures and other signal processing structures. Later, the system moves into a tuning phase and a state of continuous learning. The two stages described may also be interleaved in an iterative fashion, which calls for some kind of incremental or circular development model as will be described later.

Only very few of the most used ANN models are found in the context of continuous learning, but with minor modifications most of them can be turned into a continuous learning model.

The mapping of ANN algorithms onto highly parallel computational structures has been widely investigated. A summary is provided in [Nordström and Svensson, 1992], where processor arrays of SIMD type are pointed out as the major candidate architecture for fast general purpose neural computation.

A basic SIMD processor array is outlined in Figure 2. We have performed detailed studies of the execution of the predominant ANN models on this kind of computing structures [Gustafsson 1989, Svensson 1989, Svensson and Nordström 1990, Nordström 1991a, Nordström 1991b]. The mappings of the models and the results obtained are summarized in the subsequent subsections. A major conclusion is that broadcast or ring communication among the Processing Elements (PEs) of the array can be very efficiently utilized and actually provides the necessary means for communication within the array. Multiplication is the single most important operation in ANN computations. In bit-serial architectures, which have been our primary target, there is therefore much to gain if support for fast multiplication is added. In some of the ANN models, for example Sparse Distributed Memory (SDM), tailored hardware to support specific PE operations pays off very well.

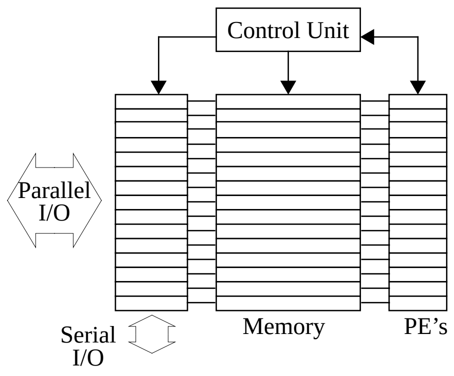

Figure 2. SIMD Module

The system architecture, described later, permits two or more modules to be linked together to form a larger module, if necessary. This linking may be done either over the communication medium, in which case the intermodule communication shares time with all modules of the system, or over a separate medium. In the latter case the cooperating modules form a *cluster* with more available bandwidth for internal communication. Special “dual-port” nodes form the interface between the cluster and the main medium.

2.1 Parallelism in ANN Computations

As described more thoroughly in [Nordström and Svensson, 1992], six different dimensions of parallelism can be identified in neural network computations: *Node parallelism* and *weight parallelism* are the two most important for consideration in a parallel implementation for use in real time. Node parallelism means treating all, or several, nodes in a layer simultaneously by several PEs. Weight parallelism means treating all, or several, inputs to a node simultaneously. The two forms of parallelisms may be combined. In typical ANN applications the degrees of these two forms of parallelism are usually very high (hundreds, thousands,...). The same, or even higher, degrees are available by *training-session* and *training-example parallelism*, but these forms are not available for use in real-time training situations, thus are of minor importance in action-oriented systems. *Layer parallelism* (treating all layers in parallel and/or going forward and backward simultaneously) and *bit-parallelism* (treating all bits in a data item in parallel) complete the picture, but the degrees of these are seldom greater than the order of ten.

In the architectures and mappings described in subsequent sessions we find it practical to refer to the different dimensions of parallelism as defined above.

2.2 Feedforward Networks with Error Backpropagation

The mapping of feedforward networks with error backpropagation on highly parallel arrays of bit-serial PEs is described in [Svensson, 1989] and [Svensson and Nordström, 1990]. Node parallelism is used. A quite simple bit-serial multiplier structure using carry-save technique [Fernström et al. 1986] is added to the basic PE design. By this, multiplication time is equalized to addition time. When performing multiply-and-add operations, which is the dominating operation in this algorithm, both units work in parallel. Connection weights are stored in matrices, one row of the matrix per PE module.

In REMAP^3, PE arrays along these lines are being developed. Figure 3 shows the design of one such PE.

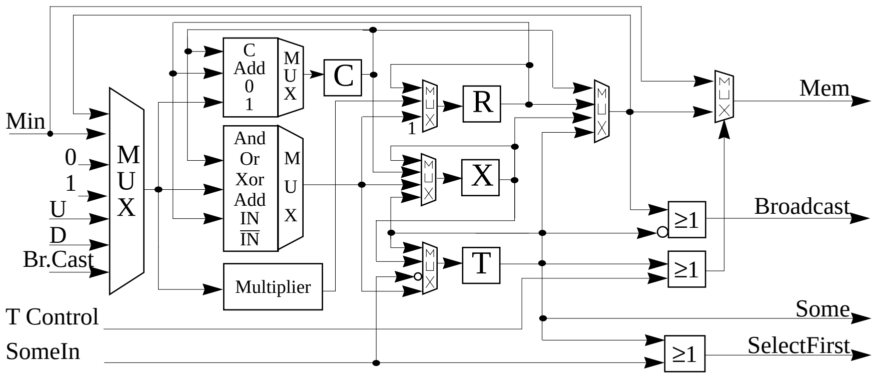

Figure 3. Sample PE from REMAP^3

An interesting result is that the computations do not require the PE array to have a very rich communication structure. The facilities needed are the ability to broadcast a single bit from any processor to all others, a means for selecting processors in order, one by one, and a bit-serial adder tree to add the values of a field. As an alternative to broadcast, ring communication may be provided; in that case the adder tree is not needed.

A typical module (about the size of one small printed-circuit board using common state-of-the-art technology) would be a 1024 PE array of bit-serial processors incorporating a bit-serial multiplier. Such an array is capable of training at 265 MCUPS (Million Connection Updates Per Second) or recall at 625 MCPS (Million Connections Per Second) using 8-bit data at 25 MHz. A four-layered feedforward network with 1024 neurons per layer would run at the speed of 85 training examples or 200 recall examples per second.

2.3 Feedback Networks

As reported in [Gustafsson, 1989] and [Svensson and Nordström, 1990], a simple PE array with broadcast or ring communication may be used efficiently also for feedback networks (Hopfield nets, Boltzmann machines, recurrent backpropagation nets, etc.). The MCPS measures are, of course, the same as above. On a 1024 PE array running at 25 MHz, 100 iterations of a 1024-byte input pattern takes 106 ms.

2.4 Self-Organizing Maps

[Nordström, 1991b] describes different ways to implement Kohonen's Self-Organizing Maps (SOMs) [Kohonen, 1990] on parallel computers. The SOM algorithm requires an input vector to be distributed to all nodes and compared to their weight vectors. This is efficiently implemented by broadcast and simple PE designs. The subsequent search for minimum is extremely efficient on bit-serial processor arrays. Determining the neighbourhood for the final update part can again be done by broadcast and distance calculations. Thus, also in this case, broadcast is sufficient as the means of communication. Node parallelism is, again, simple to utilize. Efficiency measures of more than 80% are obtained (defined as the number of operations per second divided by the maximum number of operations per second available on the computer).

2.5 Sparse Distributed Memory

Sparse Distributed Memory (SDM), developed by Kanerva [Kanerva, 1988], is a two-layer feedforward network, but is more often – and more conveniently – described as a computer memory. It has a vast address space (typically 10^{300} possible locations) which is only very sparsely (of course) populated by actual memory locations. Writing to one location influences locations in the neighbourhood (e.g. in the Hamming-distance respect) and, when reading from memory, several neighbouring locations contribute to the result.

The SDM algorithm requires distribution of the reference address, comparison and distance calculation, update, or readout and summation, of counters at the selected locations. Nordström [Nordström, 1991a] identifies the requirements for these tasks and finds a "mixed" mapping (switching between node and weight parallelism in different parts of the calculation) that is especially efficient.

A counter in the place of the multiplier in the bit-serial-PE based architecture described above makes the array especially efficient for SDM. A 256 PE REMAP^3 realization with counters is found to run SDM at a speed 10 - 30 times faster than that of an 8K PE Connection Machine CM-2 (clock frequencies equalized). Already without counters (then the PEs become extremely simple) a 256 PE REMAP^3 outperforms a 32 times larger CM-2 by a factor of 4 - 10. One explanation of this is the more developed control unit of REMAP^3 which makes the mixed mapping possible to use.

3 APPLICATION SYSTEM DEVELOPMENT

Increased flexibility, adaptability, and the potential to solve some hard problems are the main reasons for introducing ANN in real-time control systems. A new development philosophy, that allows conventional control engineering and ANN principles to be mixed, is required.

3.1 Trainability in Real-Time Control Systems

The most common development philosophy today in the domain of computer-based systems is the "sequence of phases" strategy, often referred to as the waterfall model [see, e.g., Sommerville, 1989] (Figure 4).

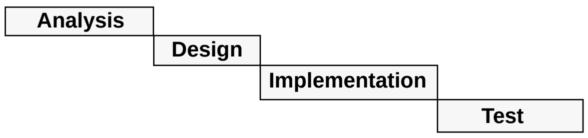

Figure 4. The waterfall model.

The sequence of phases is no longer relevant when trainable systems are to be developed. A trainable ANN system may be considered as having two parts: structure and data. The structure is the ANN algorithms and the hardware architecture. The data is the information that the system gets from the environment and the stored information that yields the behaviour of the system (e.g., the connection weights). In most of the models that have been suggested so far the structure is static in the sense that it is not changed by the system itself, but there is an interesting development going on towards dynamic structures. The stored information can be static after a training session or dynamic meaning that the environment constantly influences the system's behaviour.

In a development model feasible for trainable systems, the analysis activity has similarities to the waterfall model in sorting out the demands on the system, but turning these demands into, e.g., functions or objects is not relevant here. In contrast to programmed systems the main design task is to determine an adequate set of ANN-algorithms and a system architecture. This does not give the system its function, which is an important difference to conventional systems. The function of the system is given by training, either in a special training session or by running the system in its proper environment.

To describe development of trainable systems we need a circular development model as illustrated in Figure 5.

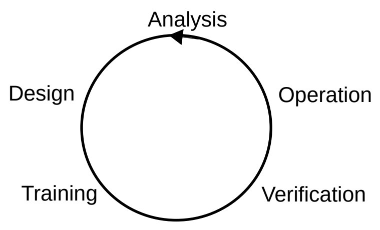

Figure 5. The circular development model.

In contrast to the waterfall model, where system development is considered as a project and maintenance as a process, the circular development model incorporates development and maintenance as two activities in the same process. The parts of this process are:.

Analysis. Each instance of this activity handles a portion of the demands that the system is to fulfil. The treated demands may have impact on the system as a whole or only a small part of it.

Design. To meet the demands, existing algorithms are tested/modified or new ones are developed. This design style can be compared to rapid prototyping to encourage the creativity of the developer. The activity leads to a structure which includes ANN-algorithms and conventional control algorithms.

Training. When the structure of the system is updated, the system is given its new properties by exposing it to environment data or a set of training data. Training may be a part of the operation activity but can also be a separate activity succeeded by verification.

Verification. In most cases the updated trained structure of the system has to be verified before letting it influence the environment. In this activity the developer can use own data or data from the environment and structures dedicated to verification.

Operation. There is no sharp distinction between operation and other activities. The behaviour of the system might change constantly during operation due to adaptation. The system might have only a fraction of its functionality implemented but still be a good test-bench for analysis, design and training.

In control applications the security aspect is often emphasized. Letting ANN-based systems act on the environment without special precautions could lead to severe problems. It is a major research challenge of neural control engineering to devise solutions for handling these matters. One possible approach is to have a "security shell" which gives limits for the outputs from the ANN algorithms.

3.2 The Continuous Development Paradigm.

To support the development model we introduce the Continuous Development Paradigm (CD-paradigm). This paradigm can be expressed as “Development by changing and adding”. This is a well-known approach in modern Software Engineering but in this context the aims are extended to include both hardware and software. A development environment to support the use of the CD-paradigm should share the following characteristics:

- Easy to change the system structure (hardware and software) and data “on the fly”.
- Incremental Development using the running system as development platform.
- No undesired side-effects on the already tested parts of the system
- System data and structures can be viewed with emphasis on understandability.
- Developer gets immediate response to a change of the system.
- Developer can use concepts and symbols of the application domain.

3.3 An Implementation

We describe an implementation of a system development tool based on the CD-paradigm described above.The most important features of the tool are:

- Graphical developer’s interface.
- Cyclic execution with temporal deterministic behaviour.
- Dynamic change of the running software.
- Dynamic inspection/change of data “on the fly”.
- Change of the distributed hardware “on the fly”.
- Use of symbols and concepts from the domain of control engineering and ANN.

The tool is used to develop applications running on a set of distributed, communicating nodes. Each node is to have a cyclically executing program. The cyclical execution scheme is chosen in order to achieve a time-deterministic behaviour. The cycles have two parts: the Monitor and the Work Process. The Monitor (i)starts on a given time (a new *dt* has passed), (ii) takes care of input data that has arrived during the previous cycle and prepares output data that is to be distributed during the present one, (iii) handles program changes, and (iv) starts the Work Process.

A temporal view of the execution of one cycle is shown in Figure 6, where the different paths of the Work Process are indicated. Continuous lines indicate processing that consumes time, dotted lines show idle processing, and lines splitting up means a selection in the control flow. The development tool guarantees that the worst case branch is within the cycle time, *dt*.

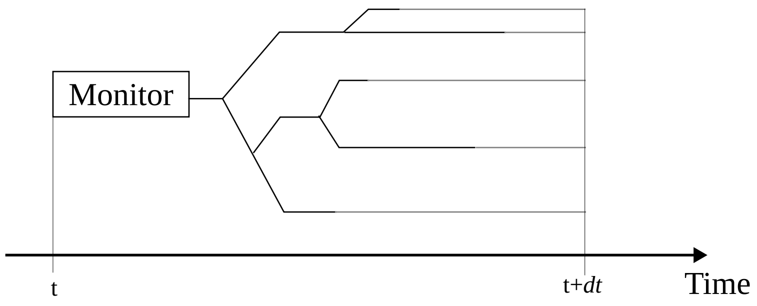

Figure 6. Temporal view of Monitor and Work Process

3.3.1 Graphical Developer’s Interface

To support the CD-paradigm and demands of understandability the developer's interface to the system is an interactive graphical tool. The most basic properties of the tool are outlined below.

- *All development is done on a system in operation.* That is, a system operating in real time but not necessarily affecting the system environment.
- *Hierarchical way of describing the application.* The levels of abstraction span from the instructions of the node control unit to the abstract concepts of the application.
- *Support for reuse of system components.* Part of the tool is a browser where system components (processes, data, and connections) are stored.
- *Tools for viewing data*. Data can be viewed in various ways, e.g. using bargraphs, diagrams, maps, and conditional recording.

On the highest level (*system level*) the user works with a display showing an overview of a typed dataflow between nodes executing cyclic processes (Figure 7). This is actually a map of the system configuration.

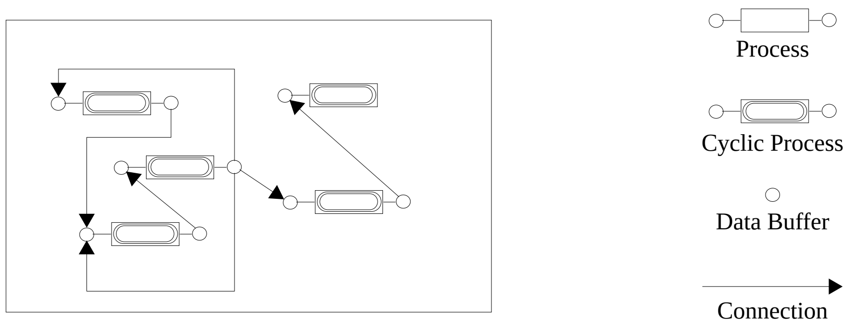

Figure 7. System level display (left) and basic symbols.

The user may open up a process symbol to work with a graphical specification on the *node level*. This can be repeated, resulting in a hierarchy of graphical specifications. Figure 8 shows an example of such a display. In the Work Area (WA), surrounded on both sides by the Input and Output areas, respectively, the designer can place symbols that specify the operation of the node. The placement of symbols in WA has temporal meaning relative to a time scale T that indicates the total time of the process. Every symbol in WA can be opened to move the designer one level of abstraction lower in the system hierarchy. When the designer places a symbol in WA, using the browser, the corresponding process will be added to the execution thread. The designer can then immediately use the inspection tools to verify the function of the added process. This is indicated in Figure 8.

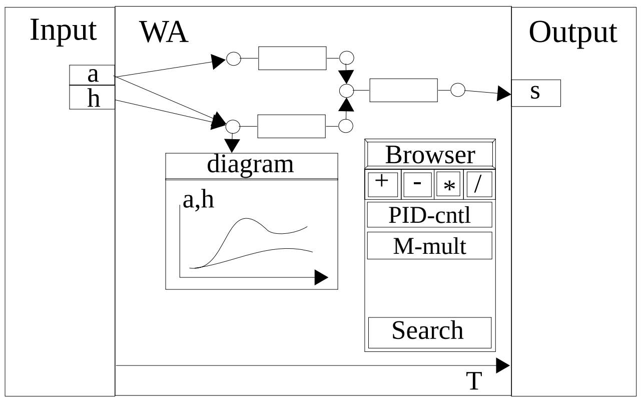

Figure 8. Node level (or lower levels) display

4 SYSTEM ARCHITECTURE AND INTERMODULE COMMUNICATION

4.1 Concept

The system-architectural concept is based on the notions of nodes, channels, and local real-time databases:

Nodes, which differ in functionality, are communicating via a *shared medium*. Input nodes deliver sensor data to the rest of the system and may perform perceptual tasks. Output nodes control actuators and may perform motoric control tasks. Processing nodes perform various kinds of calculations. I/O nodes and processing nodes may have great similarities but, because of their closeness to the environment, I/O nodes have additional circuits for interfacing to sensor and actuator signals.

Communication between nodes takes place via *channels*. A communication channel is a logical connection on the shared medium between a sending node and one or more listening nodes. The channels are statically scheduled so that the communication pattern required for the application is achieved. This is done by the designer. Two types of data are transported over the medium: *Code changes* are distributed to the nodes to allow modifications "on the fly" of the cyclically executed programs in the nodes. *Process data* informs the nodes about the status of the environment (including the states of other nodes). If the application requires intensive communication within a set of related nodes a hierarchical communication can be set up. The related nodes form a cluster with more available bandwidth on the internal channels.

Rather than being individual signals, the process data exchanged between the nodes is more like patterns, often multi-dimensional. Therefore, the shared medium must be able to carry large amounts of information (Gigabits per second in a typical system).

Every node in the system executes its program in a cyclic manner. The cyclically executed program accesses its data from a *local real-time database* (LRTDB). This LRTDB is updated, likewise cyclically, via channels from the other nodes of the system.

The principle of resource adequacy, the cyclic paradigm and the statically scheduled communication via the LRTDBs imply the time-deterministic behaviour of the system which is so important in real-time applications (cf [Lawson, 1992a]).

One of the nodes connected to the network is a Development Node, as shown in Figure 9. It establishes a channel to an executing node when it needs to send program changes. Instructions along with address information are sent to the executing node where the monitor makes the change between two executions of the Work Process.

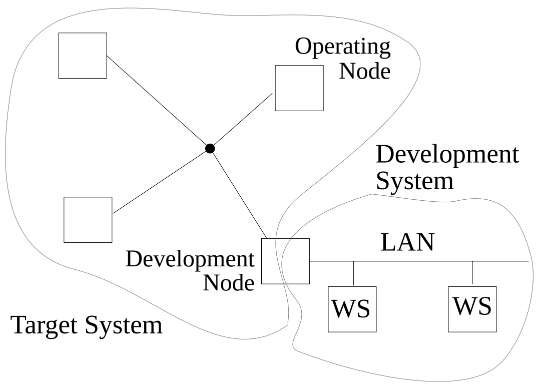

Figure 9. Multi-node target system and multiple-workstation development system

The Development Node is connected to a Local Area Network (LAN) of workstations (WS) running the development system. The LAN connection can be removed without affecting the running system.

For inspection of the LRTDB and other local data the Development Node opens channels in the same way as when other process data is moved between nodes.

4.2 Implementation

Implementations of the processing modules have been briefly described in earlier sections (see Figure 2 and Figure 3). Here we concentrate on the implementation of the communication architecture. A more detailed description is given in [Nilsson et al., 1992].

An all-optical network (the entire path between end-nodes is passive and optical) is used as the shared medium. Communication channels between SIMD Nodes are established by time-multiplexing (TDMA) in a statical manner. In every scheduled time slot there is one sender and one or more listener (broadcast).

If higher capacity is needed, WDMA (wavelength division multiple access) may be used instead. Then, scheduling of communication is not required. The nodes scan the wavelength spectrum to fill their LRTDBs. The scanning can be statically determined or a function of the internal state of the node. As an interesting future possibility, it may also be trained.

Broadcast implies that it is important to synchronize the communication. The synchronization is done via a global, distributed optical clock. Alternatively, a communication slot can be several time slots, which gives a slower communication speed.

In the communication interface of each SIMD Node (Figure 10) a clock frequency reduction is done by a factor k by means of shiftregisters (k is the size of the PE array, e.g. k=256). It is important to synchronize the dataflow with the shift clock. This is done by sending the clock and the data in the same medium. Clock and data use different wavelengths (f_1 and f_2), implying that the communication interface must include two laserdiodes and two optical filters (F) for the flow of process data.

In addition to the exchange of process data there is also a distribution of code caused by program changes made "on the fly".

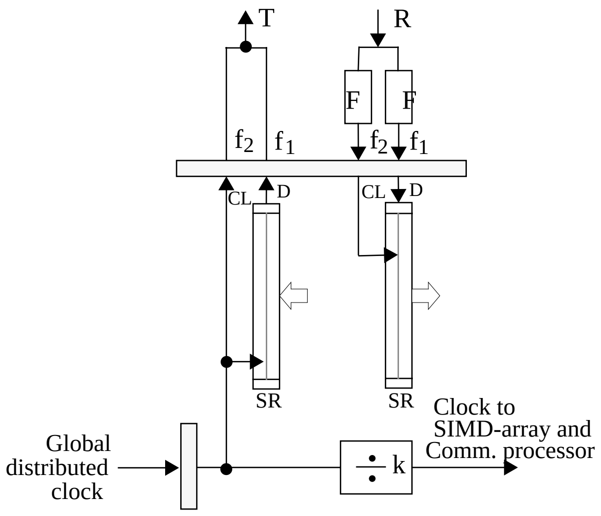

Figure 10. Communication interface. T is transmit, R is receive. Grey boxes indicate the optical/ electrical conversion.

Due to the high speed the communication interface must be integrated into one IC to work properly. Today there are shiftregisters available implemented in GaAs-technology for very high speed (some Gbit/s). The GaAs-technology also gives the possibility to integrate optical devices with logic. The topology of the all-optical network is a star, which has a decibel loss proportional to logN, while a bus topology has one proportional to N (N is the number of nodes in the system) [Green, 1991].

The SIMD Module accesses data from its own local real-time database (LRTDB) reflecting the status of the environment. The LRTDB is implemented as a dual-port memory. At one side the SIMD Module accesses data; at the other side the control unit in the communication module is updating the LRTDB via the communication interface. The control unit cyclically executes the statically scheduled send and receive commands necessary for carrying out the communication pattern of the node (Figure 11).

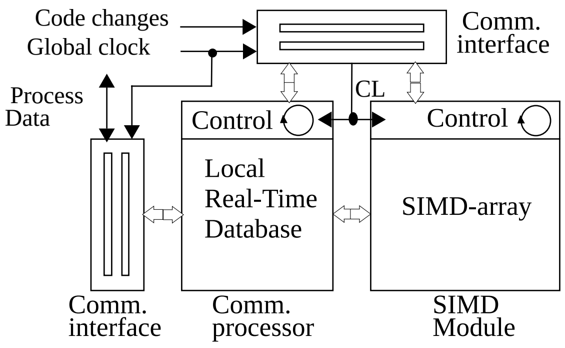

Figure 11. A Node

4.3 REMAP Prototype Development

REMAP^3 is an experimental project. A sequence of gradually evolved prototypes is being built, starting with a small, software configurable PE array module, implemented as a Master's thesis project [Linde and Taveniku, 1991]. With only slight modifications in PE array architecture, but with a new high-performance control unit, the second prototype is now being built [Bengtsson et al., 1991], almost full-scale in PE number, but far from miniaturized enough for embedded systems.

The early prototypes rely on dynamically programmable logic cell arrays (FPGAs) [Linde et al. 1992]. Therefore, different variations of the prototypes can be realized by reprogramming. The FPGAs are designed for high speed. Thus, the speed and the logical size of the prototype systems suffice for new, demanding applications, but the physical size does not allow embedded multi-module systems to be built from the prototypes.

Based on the experiences from the FPGA-based prototype modules, a design for a VLSI implemented module that can be used in multi-node systems as described above will be made.

5 CONCLUSION

This paper points to the strength of combining massively parallel architectures, trainability, and incremental development environments. The SIMD paradigm combines single-threaded programming with multiprocessing power and easy miniaturizing for embedded systems. We have presented a massively parallel system architecture based on multiple SIMD processor arrays to allow the implementation of real-time, ANN-based training using interaction-based system development tools.

The presented system architecture and development model are intended to be used in biologically inspired design of control systems [Kuperstein, 1991; Singer, 1990], where sensory, motoric, and higher cognitive functions are mapped onto nodes or clusters of nodes.

6 REFERENCES

Arbib, M.A. (1989). Schemas and neural networks for sixth generation computing. *Journal of Parallel and Distributed Computing*, Vol. 6, No. 2, pp. 185-216.

Bengtsson, L., A. Linde, T. Nordström, B. Svensson, M. Taveniku, and A. Åhlander (1991). Design and implementation of the REMAP^3 software reconfigurable SIMD parallel computer, *Fourth Swedish Workshop on Computer Systems Architecture,* Linköping, Sweden, January, 1992. Available as Research Report CDv-9105 from Centre for Computer Science, Halmstad University, Halmstad, Sweden.

Fernström, C., I. Kruzela, and B. Svensson (1986). *LUCAS Associative Array Processor – Design, Programming and Application Studies*. Vol. 216 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Green, P.E. (1991). The future of fiber-optic computer networks. *Computer*, Vol. 24, No. 9.

Gustafsson, E. (1989). A mapping of a feedback neural network onto a SIMD architecture, Research Report CDv-8901, Centre for Computer Science, Halmstad University, May 1989.

Kanerva, P. (1988). *Sparse Distributed Memory.* MIT Press. Cambridge, MA, USA.

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*. Vol. 78, No. 9. pp. 1464-1480.

Kuperstein, M. (1991). INFANT neural controller for adaptive sensory-motor coordination. *Neural Networks*, Vol. 4, pp. 131-145.

Lawson, H.W. (1992a). Cy-Clone: an approach to the engineering of resource adequate cyclic real-time systems. *The Journal of Real-Time Systems*. Vol. 4, No. 1, pp. 55-83.

Lawson, H.W. (1992b), with contributions by B. Svensson and L. Wanhammar . *Parallel Processing in Industrial Real-Time Applications*. Prentice-Hall, Englewood Cliffs, NJ, USA.

Linde, A. and M. Taveniku (1991). LUPUS – a reconfigurable prototype for a modular massively parallel SIMD computing system. Masters Thesis Report No. 1991:028 E, Division of Computer Engineering, Luleå University of Technology, Luleå, Sweden (in Swedish).

Linde, A., T. Nordström, and M. Taveniku (1992). Using FPGA to implement a reconfigurable highly parallel computer. *Second International Workshop on Field-Programmable Logic and Applications*, Vienna, Austria, Aug. 31 – Sept. 2.

Nilsson, K., B. Svensson, and P.A. Wiberg (1992). A modular, massively parallel computer architecture for trainable real-time control systems. *AARTC '92: 2nd IFAC Workshop on Algorithms and Architectures for Real-Time Control*, Seoul, Korea, Aug.31 – Sept. 2.

Nordström, T. (1991a). Sparse distributed memory simulation on REMAP^3. Research Report No. TULEA 1991:16, Luleå University of Technology, Luleå, Sweden.

Nordström, T. (1991b). Designing parallel computers for self organizing maps. Research Report No. TULEA 1991:17, Luleå University of Technology, Luleå, Sweden.

Nordström, T. and B. Svensson (1992). Using and designing massively parallel computers for artificial neural networks. *Journal of Parallel and Distributed Computing*, Vol. 14, No. 3, pp. 260-285.

Singer, W. (1990). Search for coherence: a basic principle of cortical self-organization. *Concepts in Neuroscience*, Vol. 1, No. 1, pp. 1-26.

Sommerville, I. (1989). *Software Engineering. 3rd ed.* Addison-Wesley, Reading, MA, USA.

Svensson, B. (1989). Parallel imlementation of multilayer feedforward networks with supervised learning by back-propagation, Research Report CDv-8902, Centre for Computer Science, Halmstad University, Halmstad, June 1989.

Svensson, B. and T. Nordström (1990). Execution of neural network algorithms on an array of bit-serial processors. *Proceedings of* 10*th Internatinal Conference on Pattern Recognition – Computer Architectures for Vision and Pattern Recognition*, Atlantic City, NJ, USA, June 1990, Vol. II, pp. 501-505.