DOCTORAL THESIS 1995:162 D

DIVISION OF COMPUTER SCIENCE AND ENGINEERING

ISSN: 0348 – 8373

ISRN: HLU - TH - T - - 1994 - 162 - D - - SE

Highly Parallel Computers for Artificial Neural Networks

Tomas Nordström

TEKNISKA HÖGSKOLAN I LULEÅ
LULEÅ UNIVERSITY OF TECHNOLOGY

Doctoral Thesis 1995:162 D

Highly Parallel Computers for Artificial Neural Networks

Tomas Nordström

Division of Computer Science & Engineering
Luleå University of Technology, Sweden
E-mail: tono@sm.luth.se

March 1995

Supervisor / Handledare
Professor Bertil Svensson, Chalmers University of Technology

Faculty opponent / Fakultetsopponent
Associate Professor Dan Hammerstrom, Oregon Graduate Institute

I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the seashore, and diverting myself in now and then finding a smoother pebble, or a prettier shell than ordinary whilst the great ocean of truth lay all undiscovered before me.

ISAAC NEWTON

ISSN: 0348-8373
ISRN: HLU-TH-T--162-D--SE

Published 1995
Printed in Sweden by "Högskolans Tryckeri, Luleå"

ABSTRACT

During a number of years the two fields of artificial neural networks (ANNs) and highly parallel computing have both evolved rapidly. In this thesis the possibility of combining these fields is explored, investigating the design and usage of highly parallel computers for ANN calculations.

A new system-architecture REMAP (Real-time, Embedded, Modular, Adaptive, Parallel processor) is presented as a candidate platform for future action-oriented systems. With this new system-architecture, multi-modular networks of cooperating and competing ANNs can be realized. For action-oriented systems, concepts like real-time interaction with the environment, embeddedness, and learning with self-organization are important. In this thesis the requirements for efficient mapping of ANN algorithms onto the suggested architecture are identified. This has been accomplished by studies of ANN implementations on general purpose parallel computers as well as designs of new parallel systems particularly suited to ANN computing. The suggested architecture incorporates highly parallel, communicating processing modules, each constructed as a linear SIMD (Single Instruction stream, Multiple Data stream) array, internally connected using a ring topology, but also supporting broadcast and reduction operations.

Many of the analyzed ANN models are similar in structure and can be studied in a unified context. A new superclass of ANN models called localized learning systems (LLSs) is therefore suggested and defined. A parallel computer implementation of LLSs is analyzed and the importance of the reduction operations is recognized. The study of various LLS models and other commonly used ANN models not contained in the LLS class, like the multilayer perceptron with error back-propagation, establishes REMAP modules as an excellent architecture for many different ANN models, useful in the design of action-oriented systems.

Descriptors

Action-oriented systems, artificial neural networks, sparse distributed memory, self-organizing maps, multi-layer perceptrons, localized learning systems, massively parallel computers, SIMD, bit-serial processor array, REMAP.

PREFACE

This thesis deals with the implementation of artificial neural networks on massively and highly parallel computers.

The thesis consists of nine papers. The first two papers introduce the concept of nonconforming massively parallel computers and survey parallel computer architectures used for artificial neural networks. The two following papers are descriptions of the REMAP architecture which I use as a starting point for my studies.This architecture has been developed at Luleå University of Technology, in cooperation with Chalmers University of Technology and Halmstad University. In the last five papers the mapping of different ANN algorithms onto parallel computers is studied.

As each of the papers is self-contained there are sections that overlap, but the major part of each paper is original work. The nine papers are:

PAPER A

Paper A is published in *Proceedings of the New Frontiers, a Workshop of Future Direction of Massively Parallel Processing*:

> Davis, E. W., T. Nordström and B. Svensson, "Issues and applications driving research in non-conforming massively parallel processors," in Proceedings of the New Frontiers, a Workshop of Future Direction of Massively Parallel Processing, Scherson Ed., McLean, Virginia, 1992, pp. 68-78.

PAPER B

Paper B is published in *Journal of Parallel and Distributed Computing*:

> Nordström, T. and B. Svensson, "Using and designing massively parallel computers for artificial neural networks," Journal of Parallel and Distributed Computing, vol. 14, no. 3, pp. 260-285, 1992.

PAPER C

Paper C is published in *Proceedings of Connectionism in a broad perspective*:

> Svensson, B., T. Nordström, K. Nilsson and P.-A. Wiberg, "Towards modular, massively parallel neural computers," Connectionism in a Broad Perspective: Selected Papers from the Swedish Conference on Connectionism - 1992, L. F. Niklasson and M. B. Bodén Eds. Ellis Horwood, pp. 213-226, 1994.

PAPER D

Paper D is published in *Selected papers from: Second International Workshop on Field-Programmable Logic and Applications*:

> Linde, A., T. Nordström and M. Taveniku, "Using FPGAs to implement a reconfigurable highly parallel computer," Field-Programmable Gate Array: Architectures and Tools for Rapid Prototyping; Selected papers from: Second International Workshop on Field-Programmable Logic and Applications (FPL'92), Vienna, Austria, Grünbacher and Hartenstein Eds. New York: Springer-Verlag, pp. 199-210, 1992.

PAPER E and F

Paper E and F are submitted for publication:

> Nordström, T., "On-line localized learning systems, part I - model description," submitted for publication, 1995.
>
> Nordström, T., "On-line localized learning systems, part II - parallel computer implementation," submitted for publication, 1995.

These papers are also available as research reports:

> Nordström, T., "On-line localized learning systems, part I - model description," Res. Rep. TULEA 1995:01, Luleå University of Technology, Sweden, 1995.
>
> Nordström, T., "On-line localized learning systems, part II - parallel computer implementation," Res. Rep. TULEA 1995:02, Luleå University of Technology, Sweden, 1995.

PAPER G

Paper G is to be submitted for publication:

> Nordström, T., "Hardware for sparse distributed memory simulations," to be submitted for publication, 1995.

An earlier version is available as a research report:

> Nordström, T., "Sparse distributed memory simulation on REMAP3," Res. Rep. TULEA 1991:16, Luleå University of Technology, Sweden, 1991.

PAPER H

Paper H was presented at *Fourth Swedish Workshop on Computer System Architecture*:

> Nordström, T., "Designing parallel computers for self organizing maps," in DSA-92, Fourth Swedish Workshop on Computer System Architecture, Linköping, Sweden, 1992.

This paper is also available as a research report:

> Nordström, T., "Designing parallel computers for self or ganizing maps," Res. Rep. TULEA 1991:17, Luleå University of Technology, Sweden, 1991.

PAPER J

Paper J is published in *Proceedings of 10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*:

> Svensson, B. and T. Nordström, "Execution of neural network algorithms on an array of bit-serial processors," in 10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition, Atlantic City, New Jersey, USA, 1990, vol. II, pp. 501-505.

THESIS SUMMARY

Motivation and significance

Both the computer and the model of biological neural networks emerged during the ‘40s. Many computer architects have been inspired by how the human brain works. Already von Neumann [22] discussed the relation between human and machine computations. Many neural-like algorithms or machines emerged during the late ‘50s and early ‘60s, and were used in the fields of pattern recognition, classification and adaptive signal processing. However, after the book “Perceptrons” by Minsky and Papert appeared in 1969 the artificial neural network field seemed to be a dead end. In their book Minsky and Papert showed the limitations of the most popular model at that time: the perceptron. The results in the book were correct and elegantly described, but the authors also speculated that more complex models, e.g. multi-layer ones, would show the same limitations as the simple perceptrons, a speculation which was later proven to be wrong. In the ‘70s and early ‘80s much of the research on human-like capabilities for computers was conducted as research in the field of artificial intelligence using a symbol oriented approach, in which Minsky and Papert became prominent.

In parallel with the research on human-like capabilities for computers a continuous evolution of computer hardware took place. Technological development in the fields of semiconductors, transistors, and integrated circuits resulted in an enormous increase in calculation speed. Most of the computers before the ‘80s were uniprocessors or several loosely connected individual computers. Despite the power of the computer in the early ‘80s, there seemed to be little progress in the solving of problems like image recognition, speech recognition etc. It became apparent that new algorithms and computing models were needed to solve these problems, which humans solve without effort.

During the early ‘80s some of the dedicated scientists who had continued their effort after the publishing of Minsky’s and Papert’s book, like Kohonen, Grossberg, Anderson, Rumelhart and McClelland together with some new scientists in the field like Hopfield, gradually built a foundation on which many new and powerful artificial neural network models could be built. In the late ‘80s the field of artificial neural networks literally exploded and efforts from many researchers from biology, physics, mathematics, control theory, psychology, computer science, and computer engineering demonstrated the capabilities and possibilities of the artificial neural network models. At the same time a number of commercial, massively parallel computers were manufactured. As many of the new neural network models are massively parallel by nature they seem to map very well onto these new massively parallel computers.

However, in many instances the current high performance, parallel, general purpose computers are not as well suited as they first appear to be since they do not address issues like real-time, time determinism, heterogeneous communication, physical size, or power consumption. These issues are important in the realization of *action-oriented systems* [1, 2]

which interact in real-time 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. These systems will be trainable in contrast to the programming of today's computers. All these issues can be viewed as *nonconforming* to the systems available on the more general purpose oriented market for parallel computers.

The common theme in this thesis is the idea of finding an architecture suitable with respect to these issues and thus suitable for action-oriented systems. A new system architecture, REMAP (Real-time, Embedded, Modular, Adaptive, Parallel processor project), incorporating highly parallel, communicating processing modules, is presented as a candidate platform for future multi-modular artificial neural networks (ANNs). Action-oriented systems are studied mainly by focusing on separate ANN modules (algorithms) and on separate hardware modules, but all these software and hardware modules are parts in the concept of a modular and heterogeneous system.

Approach

For this thesis the approach has been to start with simple processing elements and simple means of communication. In a way this approach is inspired by the relatively simple building blocks (nerve cells) the brain consists of. As needed, architectural features have been added to this basic concept. For the major part of my study the REMAP architecture has been used as a starting point. This is a reconfigurable bit-serial SIMD (Single Instruction stream, Multiple Data stream) processor array . As the processing elements are reconfigurable it is possible to include dif ferent types of support for dif ferent kinds of algorithms. Architectural principles and components that are essential for the ef ficient simulation of the most frequently used ANN models are established in the thesis. One of the results is that surprisingly few additions to the basic concept are required to get good performance and high efficiency.

Summary Outline

The rest of this thesis summary is outlined as follows: first I discuss the design space on a general level; then follows a more detailed discussion of the design space for control, processing elements (PEs), and the interconnection network. The next major section will discuss the ANN implementation studies I have done. In the sub-sections some of the findings which make it possible to find a suitable architecture in the vast design space will be pointed out. These findings are used in the following section to design our own REMAP architecture and the REMAP prototype. After that I briefly point out possible future directions for this research. The major conclusions of this thesis are then summarized in the next section. Finally there are sections with corrections and comments, abbreviation list, acknowledgments, and references.

Design space

The design space for computers that can be used for ANN is very large, and even if it seems futile to talk about an optimal design there are a number of interesting trade-offs to be made. One basic trade-off is between *flexibility* and *speed*. That is, to increase speed we usually have to sacrifice some flexibility. In one end of the spectrum we have ordinary general purpose computers and in the other end we might have analog or optical computers. In between we have various digital parallel computer designs.

The large number of ANN algorithms and the continuous development of new models and variations suggest that a certain degree of flexibility through reprogramming is desirable. This is usually solved by constructing a general architecture that can be software programmable. With the arrival of reconfigurable hardware, such as field programmable gate arrays (FPGAs), it is possible to achieve some of the needed flexibility through "soft hardware." The possibility to build a computer for ANN with FPGAs has been explored in Paper D.

While analog or optical computers in certain situations might give higher performance than corresponding digital computers they will need to be designed for a specific algorithm and cannot easily be reprogrammed or reconfigured. Even though I can see a future where analog modules fit into the concept of multi-modular computers (esp. close to sensors) it has not been considered in this thesis. Instead I have concentrated on the part of design space where *digital*, *software programmable*, *parallel computers* can be found, as the flexibility to run many different ANN models is desirable.

Given that we want to design a digital parallel computer suitable for ANN there are still many trade-offs. The main questions are:

- What form and level of execution autonomy should the PEs have?
- What is a suitable size and complexity of the PEs? How many PEs should be available and what amount of memory should each PE have?
- How should the PEs be interconnected?

These questions correspond to the building blocks of a parallel computer: the *controller*, the *processing elements* and their memory, and the *interconnection network*, which will be discussed in the following sub-sections.

How will this (still large) design space be further reduced? The most important aspect has been the intended application: action-oriented systems, that is, be able to run a number of different artificial neural networks models.

Control design space

There has been a long controversy in the parallel computer field between the concept of *SIMD* (Single Instruction stream, Multiple Data streams) and *MIMD* (Multiple Instruction streams, Multiple Data streams) from the taxonomy by Flynn [10]. The core of the controversy is the question of how much autonomy each PE should have. Whereas the SIMD computer has one single controller the MIMD computer will use a controller for each PE, providing the PEs maximal autonomy.

Disadvantages of the MIMD concept are [5]: the problem to program and debug hundreds of processors running independently, more parallelism is found in data than in instructions, and the high cost of synchronization. A major advantage of MIMD is that highly optimized "off-the-shelf" components like RISC processors and high density memory modules can be used as building blocks.

One trend among today's general purpose parallel computers is to either extend the SIMD concept with a suitable amount of autonomy (like local address modification), or to reduce the MIMD concept to SPMD (Same Program on Multiple Data streams).

This debate can be extended to the two major paradigms used to express parallelism: *data parallelism* and *control parallelism* [5]. Whereas the former finds parallelism in the data set the latter will find parallelism in the instructions. In a study by Fox [11] he reports that the most successfully implemented problems on parallel computers were those which could be specified as data parallel.

In this thesis the basic SIMD concept, thus, a data parallel paradigm, has been found sufficient for most of the ANN models studied. However, action-oriented systems with their structure of cooperating modules suggest that the SIMD concept should be extended to a *MIMSIMD* (Multiple Instruction streams for Multiple SIMD arrays) as will be discussed in Papers A, B, C, and G.

PE design space

In the PE design space there are important decisions to be made on how many PEs there should be, and the size and complexity of these PEs. There is a trade-of f between having many simple PEs and few complex PEs, a PE *granularity* design choice.

While discussing parallel computers it is interesting to use an adjective to indicate the number of PEs available. And while studying parallel computers for ANN we found that the term *massively parallel* was used without defining what was meant by massive. Therefore we set out to make a suitable definition, thus, in Paper B we argue that being massive should mean that the structure gives an impression of being solid. That is, the number of units should be so high that it is impossible to treat them individually; they must be treated *en masse*. By definition then, each PE must be told what to do without specifying it individually. This is actually the concept of SIMD or SPMD. The lower bound for massive parallelism will then be set by the largest computer in which each PE is treated as an individual with its own instruction flow (MIMD). We think that for the moment $2^{12} = 4096$ is a suitable limit. It is useful to have characterizations also of the lower degrees of parallelism and to this end we suggest a rough division between: *highly* parallel, *moderately* parallel, and *barely* parallel. By defining the limits to 2^{12}, 2^8, 2^4, 2^0 we have an easy-to-remember scheme for the characterization. In the previous section we introduced the MIMSIMD concept as a suitable way of extending the SIMD concept for action-oriented systems, this also implies that we prefer a system of *highly parallel modules* to a single massively parallel computer.

To guide the decision on the PE complexity required it is interesting to determine the *precision* needed in calculations and for input/output (I/O). This precision is of course determined by the intended application area, ANN. From the discussion in Paper B it is clear that most ANN implementations will not need very high precision calculations (typically less than 16 bits). In general the precision used by I/O will depend on the environment in which the ANN or the action oriented system is used, but it can be noted that layers close to sensors often use low precision, say 16 bits or less.

The PE granularity decision also depends on the amount of memory needed for each PE and whether the memory should be on-chip or of f-chip. Given additional levels of autonomy, other sets of constraints might become important. For example, if local address modification is wanted, each PE's address lines need to be sent outside of the chip (assuming of f-chip memory). Thus, the number of pins becomes a limiting factor.

Many *realization aspects* will also influence the PE granularity decision. One such realization aspect is the type of VLSI (very large scale integration) technology available. Different technologies, for example CMOS, GaAs, and FPGA, will introduce dif ferent optima in our design space. Another realization aspect is packaging. That is, the number of pins that can be used to connect the PEs to external memory and I/O are limited. The load they can take and their maximum switching rate are also limited. The use of standard components, like SRAM or DRAM is desirable as such components use highly optimized realizations. Even though I, as mentioned earlier, suggest that highly parallel computer modules are better than massively parallel computers for ANN calculations, the number of PEs in a module are still too high for single chip solutions to be feasible. This requires us to have many sig-

nals to go off-chip. Within the REMAP project multi chip modules (MCM) are investigated as a possible remedy for the off-chip connectivity problem.

If a bit-parallel approach is not taken for granted, a *bit-serial* approach opens up a new set of opportunities. Besides allowing many more PEs into each chip the bit-serial approach will allow a trade-off between speed and precision, even dynamically. This lets us make important experiments on the precision needed for different ANN models. Bit-serial design also allows for *rapid prototyping* using FPGAs. Even if FPGAs have low density compared to full custom VLSI a highly parallel computer can be built with FPGAs if bit-serial PEs are chosen, as shown in Paper D.

The simplicity of bit-serial PEs allows a high clock frequency to be used. The chip area is also more efficiently used for bit-serial calculations. It can in fact be argued that bit-serial computation is always the most effective way of computing [12]. Still, for bit-serial arithmetic to be competitive in modern CMOS technology the design either needs to implement very many (more than 128) PEs or to clock them at high clock speeds (probably above 500 MHz). The possibility of such sizes and clock-speeds are shown in the Blitzen project [6] which 1990 resulted in a chip with 128 PEs in 1.25 µm standard CMOS process and by Larsson-Edefors who built a 470 MHz bit-serial arithmetic unit in 1.0 µm standard CMOS process [14]. However, there are drawbacks in having very many PEs and using high clock-speed. One problem is the size mismatch between PE array size and problem size in the case of very many PEs. For the high speed PEs it becomes hard to construct the fine-grain controller. Thus, for future CMOS implementations these problems must be addressed. However, the modular concept and a distributed (on-chip) control seem to alleviate the problems. Another possible drawback with bit-serial computation is the difficulty of performing floating point calculations, but studies by Åhlander [24] indicate that bit-serial floating point hardware is a feasible alternative.

The synaptic processing rate argument

While discussing granularity it is also interesting to discuss the balance between the processing power and the network size (that is, the number of weights). Holler [13] has introduced the concept of *synaptic processing rate* (SPR) (or CPSPW – connections per second per weight). His argument for the importance of this measure is the following: Biological neurons fire approximately 100 times per second. This implies that each of the synapses processes signals at a rate of about 100 per second, hence the synaptic processing rate (SPR) (or, to use Holler's terminology, the CPSPW) is approximately 100.

If we are satisfied with the performance of biological systems (in fact, we are even impressed by them) this number could be taken as a guide for ANN implementations. Many parallel implementations have SPR numbers which are orders of magnitude greater than 100, hence have too much processing power per weight. A conventional sequential computer, on the other hand, has an SPR number of approximately 1 (if the network has about a million synapses), that is, it is computationally under-balanced. Bit-serial PEs might be a suitable compromise between these extremes. It should be noted that Holler's argument is of course not valid in batch processing training situations.

Conclusion

Without knowing all the realization aspects there is no definite answer about the best PE granularity. But following our design approach we started to explore the simplest form of PEs, the bit-serial ones. This choice for our first prototype architecture has made it possible to make a highly parallel prototype with large flexibility by using FPGAs and the possibility to easily experiment with different precision in the calculations.

Interconnection network design space

A large number of interconnection networks (ICNs) have been suggested for parallel computers over time, thus the ICN design space is large. Even after restricting ourselves to communication networks suitable for SIMD control, that is, synchronous communication, there are many choices of topology available. The most powerful communication is found in an all-to-all network, but the cost to implement this structure quickly becomes prohibitive. At the other end of the cost/complexity scale we find the bus. Even if the structure is simple, the possibility to perform broadcast makes a bus interesting for ANN computations. Between these two extremes many topologies have been suggested [8], for example, ring, star , tree, mesh, hypercube, shuffle-exchange, omega, Benes.

The trend among highly parallel computers today is to use *low-dimensional networks* like ring, mesh, torus, or 3D networks, as well as trees and fat-trees. Each of these networks can be shown to be optimal under certain important criteria, like hardware volume or wiring cost [7, 17].

The special requirements for ANN computations make other *more specialized communication structures* attractive. For instance the use of broadcast becomes very interesting, supporting the spreading of activation values among nodes.

Design space conclusion

Based on earlier studies of highly parallel architectures [9] and a general conception of the architecture's usefulness the hypothesis has been that a *linear SIMD array*, internally connected using a *ring* topology, but also supporting *broadcast*, is a suitable architecture for ANN calculations. In this thesis I have conducted studies around this concept which, as it turns out, fits very well to the requirements of ANN calculations. As discussed in the next section two important extensions are the support for reduction operations and the support for fast multiplication.

On a higher level the REMAP concept also includes the idea to support multi-modular ANNs by using a number of these *highly parallel SIMD array modules* interconnected to form a *modular* and *heterogeneous system*.

Studies of ANN implementations on highly parallel computers

General studies of ANN implementations on highly parallel computers have been conducted as well as specialized studies of our experimental architecture REMAP . The results have then been used to restrict the hardware design space (finding what extensions to the basic concept that are needed).

Paper B surveys the most used models and describes some basics of ANNs. The computational and communication needs are analyzed for the basic models. The different dimensions of parallelism in ANN computing are identified, and the possibilities for mapping onto the structures of dif ferent parallel architectures are analyzed. Some means to measure the performance of ANN simulations are given. A survey of 27 different parallel computers used for ANN simulations is also given.

In Paper E I introduce the concept of *localized learning systems* (LLSs). This concept makes it possible to combine many commonly used ANN models into a single "superclass." The LLS model is a feedforward network using an expanded representation with more nodes in the hidden layer than in the input or output layers. The main characteristics of the model are local activity and localized learning in active nodes. Some of the well known ANN models that are contained in LLS are generalized radial basis functions (GRBF), self-or ganizing maps (SOM) and learning vector quantization (L VQ), restricted Coulomb ener gy (RCE), probabilistic neural network, sparse distributed memory (SDM), and cerebellar model arithmetic computer (CMAC). The connection between these models as variations of the LLS model is demonstrated. This connection also lets us suggest new variants of "old" ANN models. In two separate papers, Paper G and Paper H, two of the LLS models are studied in greater detail (*SDM* respectively *SOM*). Furthermore, in Paper J the mapping of two well known ANNs not contained in the LLS class, the *multilayer perceptron* (MLP) with error back-propagation and the *Hopfield network*, are studied. Thus, it covers both the mapping of feedforward and feedback neural nets. The characteristics of these models are briefly outlined. The computations are analyzed for all these models, performance figures are given, and system implementation is discussed.

Mapping ANN onto the computer architecture

Before the best mapping can be found it is important to identify the kind of parallelism found in the algorithm. In Paper B the different *dimensions of parallelism* typically found in ANN algorithms are identified as follows:

Training session parallelism
Training example parallelism
Layer and Forward-Backward parallelism
Node (neuron) parallelism
Weight (synapse) parallelism
Bit parallelism

Among these forms of parallelism the greatest amount of parallelism is found in training session, training example, node, and weight parallelism. As the first two are batch oriented and cannot be used in real-time the most interesting forms of parallelism are found to be *node and weight parallelism*. Maybe node parallelism is the most natural form, that is, mapping a node (neuron) onto a PE. This is also suggested as the basic mapping for all the models studied. However, this basic mapping becomes inefficient if there is a large mismatch between the sizes of different node layers. In Paper F a strict node parallel mapping is therefore relaxed and a weight parallel mapping is used for the final layer, thus we get a "*mixed parallelism*" solution. To support weight parallelism, facilities to combine the "weight output" is needed, thus the addition of an *adder-tree* to the architecture is suggested. This adder-tree can also support the implementation of MLPs with back-propagation learning as suggested in Paper J.

For Kanerva's SDM model described in Paper G, a node parallel mapping used for all layers is found to be the most efficient one. This is accomplished by using a so called transposed mapping.

Operations required for ANN calculations

The basic operations for ANN calculations are addition, subtraction, and multiplication. Multiplication is the most complex of these operations and can easily become the bottleneck. Therefore the *multiplication* should be supported with extra hardware.

While studying the implementation of SDM in Paper G, it was found beneficial to add a *bit-counter* to the architecture. By adding such a counter to each PE, the selection time (the first layer) can be reduced by a factor of three to four. As the basic SDM model does not use a the multiplier it could actually be exchanged for the counter. This possibility is particularly interesting if reprogrammable logic is used.

Much of the computing power of ANN comes from the use of non-linear functions, for example, the sigmoid or a Gaussian. Since the calculation of such functions can be time consuming the support for such functions needs some consideration. This is discussed in Papers B and E.

Adding control autonomy to allow addition and subtraction to take place at the same time (depending on an add/sub control bit) has been found to speed up certain parts of the SDM calculations by a factor of two.

Many of the ANN implementations studied can benefit numerically from having *saturation arithmetic* where overflow saturates to the maximum value and underflow is set to the minimum value.

Communication structure required by ANN

The basic building block of the brain is the nerve cell (neuron). In humans there are about 10^{12} neurons. But the complexity of the brain is not limited to the vast number of neurons. There is an even larger number of connections between neurons. One estimate is that there are a thousand connections per neuron on the average, giving a total of 10^{15} connections in the brain. Neurons are often grouped naturally into larger structures (hundreds of thousands of neurons). Inside these cortical areas or modules the interconnection is denser than between modules. Much of the impressing performance of biological systems comes from the highly interconnected (and modular) structure. Thus, communication is essential for the neural networks field and this is also reflected in the usage of the term connectionism. This structure should also be taken as a guide for ANN implementations.

One such multi-modular structure is suggested for *action-oriented systems*. These systems consist of a number of cooperating, often different, ANN models. These separate ANN models often use *dense interconnection patterns*. In Paper B we find broadcast or ring communication to be very efficient ways to support these dense communication patterns. It is also noted that the synchronous form of communication resulting from a SIMD control helps to achieve an effective solution. To support ANN models which use competitive learning, and which use the "*transposed mapping*" (i.e., SDM), we early on added a select-first network and global-or into the communication structure. These structures are also generally needed for general purpose programming on SIMD computers.

Later in my studies the importance of the *reduction operations* is noted, cf. Paper F. This is especially true in on-line situations where batch training is not suitable. The importance is clearly shown for LLSs in Paper F but these operations are also useful for MLP with back-propagation, even if this is not stressed in the earlier Papers B and J. Surveying other architectures (see Papers B and F) few architectures are found that have support for reduction operations, thus few of the architectures include a suitable support for the LLS class of models.

To support the communication needed between different modules an optical network connected as a star is suggested. This is briefly described in Paper C. By using time multiplexing (TDMA) this network can support real-time and time-determinism. In each module a real-time database reflects the status of the environment which is cyclically distributed over the optical network. A more thorough treatment of the intermodule communication structure is found in [18].

Real-time, Embedded, Modular, Adaptive, Parallel processor project – REMAP

The REMAP project started as "Reconfigurable, Embedded, Massively Parallel Processor Project" but has evolved into "Real-time, Embedded, Modular, Adaptive, Parallel processor project." This change reflects the change of focus in the project. The *reconfigurability* became less emphasized, even if it is still there. As discussed in Paper A we did instead find *modularity*, *adaptability* and *real-time operations* important for this type of massively parallel computer. It also has become apparent that highly parallel modules fit the intended application area better than a monolithic massively parallel computer.

The REMAP project is described in Paper C and the first realization of a REMAP module, a reconfigurable bit-serial processor array with SIMD control, is described in Paper D. To allow real-time operations and time determinism the architectural concept is based on resource adequacy, both in processing and communication [15, 16]. Learning algorithms are cyclically executed in distributed SIMD-nodes, which access their data from local real-time databases, updated with data from the other nodes via a shared high-speed optical link. Other aspects of the project are described in [3, 4, 18, 19, 21, 23]. The support for the multi-modular REMAP is an ongoing research.

As described earlier in this summary support for fast multiplication is desired. In Paper J the very simple bit-serial multiplier suggested by Ohlsson in [20] was added to the architecture. This simple *bit-serial multiplier* uses a carry-save technique and can equalize multiplication time relative to addition time. In Paper G I suggest that the bit-serial multiplier be replaced by a simple bit counter that can support fast Hamming distance calculations. For the LLS model described in Paper E I found a need for reduction operations. The architectural support for this is analyzed in Paper F. Three different implementations of *global-sum* are identified and studied in Paper F. It is found that a *bit-serial tree of adders* gives the best performance/size ratio. For the *global-minimum* operation a new bit-serial structure is proposed. This new min/max network has the advantage of not needing a global-or network which the standard bit-serial way of finding minimum needs. This also results in a speed advantage in most cases. These reduction operations can be seen as special communication structures, but besides them I have not found any other means of communication necessary in a single module.

All the studied ANN implementations in this thesis show that the REMAP concept is an excellent architecture. Minor additions have been needed, but as they have been added a high performing and efficient architecture has emerged.

The REMAP prototype

Building *custom computers* with FPGAs is today a field of research in its own right. In Paper D the possibility to build a REMAP module with FPGAs was explored. One of the findings was that in order to use the FPGA circuits efficiently, and get high performance, the signal flow is crucial. Unfortunately the Xilinx EDA software did not support this design issue at the time of implementation (1992-1993), and the signal flow design had to be made by hand. The need to assign the I/O pins to memory and controller further restricted the reconfigurability. Thus, even if the processing elements are simple and regular, which makes it easy to implement them with the XACT Editor, the possibility to reconfigure has not been used much in our project. On the other hand the implemented PE design fulfills the requirements for most ANN models and thus the need to change the PEs is limited. This design method also gives the PEs high performance, with clock rates up to 40-50 MHz. These issues are also discussed by Linde and Taveniku in [21].

The positive side of using FPGAs is that they allow you to think of the computer as "modeling clay" and you feel free to really change the architecture towards the application and not the other way around. With better tools this kind of architecture also has the potential to allow different architectural variations to be easily tested and evaluated on real applications.

Future REMAP Research

The main objective for future REMAP research is to develop the multi-modular concept. The design and use of multi-modular ANN, and the question of how to map them onto multi-modular highly parallel computers, should be addressed.

For the next generation of REMAP computers there are of course also many other research issues to be resolved. Whether a new realization technology will result in a different PE granularity is one such issue. And if the bit-serial design is retained it is clear that the control issues must be addressed.

MAIN CONTRIBUTIONS

The main results presented in this work are the following:

- A new system-architecture REMAP (Real-time, Embedded, Modular, Adaptive, Parallel processor project), incorporating highly parallel, communicating processing modules, is presented as a candidate platform for future multi-modular artificial neural networks (ANNs).
- The suggested architecture for the modules is a linear SIMD (Single Instruction stream, Multiple Data stream) array , internally connected using a ring topology , but also supporting broadcast and reduction operations. Besides addition and subtraction the PEs need to support fast multiplication, which is identified as the most important operation in ANN implementations. For certain ANN models minor additions or replacements of PE features have been suggested, for example, the addition of a bit-counter to support Kanerva's sparse distributed memory (SDM) is proposed.
- Requirements for efficient mapping of ANN algorithms onto highly parallel computer modules are identified. This has been accomplished both by studies of ANN implementations on general purpose parallel computers as well as designs of new parallel systems tuned for ANN computing.
- The terms massively/highly/moderately/barely parallel are defined. The analysis suggest that modules of highly parallel modules are better than a single monolithic massively parallel computer, i.e., each module should have less than 4096 PEs.
- Six different dimensions of parallelism in ANN calculations have been identified. For action-oriented systems it is established that node and weight parallelism are the most important. In this thesis I also establish that the combination of node and weight parallelism, "mixed parallelism", is the preferred form of parallelism for most of the ANN models studied.
- Many of the analyzed models are similar in structure and can be studied in one context. I therefore suggest and define a new superclass of ANN models called localized learning systems (LLSs). A parallel computer implementation of LLS is analyzed and the importance of the reduction operations is recognized. After adding support for reduction operations to the REMAP computer concept, it becomes very well suited for LLS.
- For many ANN algorithms, the reduction operations have been found to be an important extension to the basic SIMD architecture. Three implementations of global-sum are identified and studied. It is found that a bit-serial tree of adders gives the best performance/size ratio. For the global-minimum operation a new bit-serial structure is proposed. This new min/max network has the advantage of not needing a global-or network as the standard bit-serial way of finding minimum does. This results in a speed advantage in most cases.

- A REMAP prototype containing 128 bit-serial PEs has been built, showing that it is possible to build a highly parallel computer module suitable for ANN calculations with field programmable gate arrays.
- The potential of the suggested architecture is shown through theoretical studies, for example, an SDM implementation of a normal problem using 256 REMAP processing elements is found to run 10 times faster than the normal Connection Machine simulation, where 8k processing elements are used. Another example is the very efficient implementation of self-organizing maps on the REMAP architecture.

CORRECTIONS AND COMMENTS

This thesis consists of papers that have been published as separate documents and no corrections have been made to the content. However, there are some corrections and clarifications that should be made. These are listed below.

- The number of layers defined in Paper J is not consistent with the other papers. In Paper J the layer count is the number of *node layers*, while in Paper A to Paper H it is the number of *weight layers*.
- In Paper J on page II-502 in Algorithm 1, Step 2, replace ‘ $\text{net}_j^{(l)} = \sum_i w_{ij}^{(l)} o_i^{(l-1)}$, with ‘ $\text{net}_j^{(l)} = \sum_i w_{ji}^{(l)} o_i^{(l-1)}$,
- In Paper J the description of the problem associated with accessing the weight matrix W during the feedforward phase and needing to access the transposed matrix W^T during the back-propagation phase is sketchy. Solutions to this problem can be found in Paper B, for example using skewed matrices or an adder -tree. Additionally, three different forms of adder-trees are suggested and analyzed in Paper F.

ABBREVIATIONS

The following abbreviations are used in this thesis:

ALU	arithmetic and logic unit
ANN	artificial neural network
ANS	artificial neural systems
AOS	action oriented system
BAP	bit-serial array processor
BP	back-propagation
CL	competitive learning
CLB	combinatorial logic blocks
CMAC	cerebellar model arithmetic computer
CMOS	complementary metal oxide silicon
CPS	connections per second
CPSPW	connections per second per weight
CU	control units
CUPS	connection updates per second
DRAM	dynamic RAM
EBF	elliptic basis function
ER	expanded representation
FA	full adders
FIFO	first in first out
FLOPS	floating point operations per second
FPGA	field programmable logic arrays
GRBF	generalized radial basis function
GaAs	gallium arsenide
HD	Hamming distance
HEP	high energy physics
HyperBF	hyper basis function
I/O	input and output
ICN	interconnection network
IOB	input-output blocks
IPS	interconnections per second
LAN	local area network
LHC	large hadron collider
LLS	localized learning system
LRTDB	local real-time database
LVQ	learning vector quantization
MCM	multi-chip modules
MD	Mahalanobis distance
MIMD	multiple instruction streams, multiple data streams
MIMSIMD	multiple instruction streams for multiple SIMD arrays
MLP	multilayer perceptrons
MM	memory modules
MSB	most significant bit

PCA	principal component analysis
PE	processing element
PNN	probabilistic neural networks
RAM	random access memory; a better term would be read and write memory (RWM)
RAN	resource-allocation network
RBF	radial basis function
RBP	recurrent back-propagation
RCE	restricted Coulomb energy
REMAP	real-time, embedded, modular, adaptive, parallel processor project
RISC	reduced instruction set computer
RPCL	rival penalized competitive learning
RWC	real world computing
SDM	sparse distributed memory
SIMD	single instruction stream, multiple data streams
SOFM	self-organizing feature maps
SOM	self-organizing maps
SONN	self-organizing neural network
SPMD	same program on multiple data streams
SPR	synaptic processing rate
SRAM	static RAM
SSC	superconducting super collider
TDMA	time division multiple access
TFM	topological feature maps
TLU	threshold logic unit
TRD	transition radiation detector
VLSI	very large scale integration
WA	work area
WDMA	wavelength division multiple access
WS	workstations
WTA	winner-take-all
WUPS	weight updates per second

ACKNOWLEDGEMENTS

First and most I would like to thank my supervisor and friend Prof. Bertil Svensson, now at Chalmers University of Technology, for introducing me to the field of parallel computers, for his guidance and for his comments during the evolution of this thesis.

I would also like to send my thanks to all the co-authors of the papers in this thesis.

Furthermore, I would like to thank all the people at Luleå University of Technology who have assisted me by reading parts of this thesis and offering me their suggestions, comments, criticism, and encouragement: Prof. Per-Ola Börjesson, Prof. Svante Carlsson, Assoc. Prof. Lennart Gustafsson, Assoc. Prof. Timo Koski, Lic. Tech. Per Ödling. The support from the head of the Computer Science and Electrical Engineering department Assoc. Prof. Anders Grennberg and the head of the Computer Science and Engineering division Assoc. Prof. Lennart Andersson is much appreciated.

Discussions among the project group members of REMAP have been of the utmost value for understanding the possibilities of the REMAP computer architecture.

I also would like to acknowledge Prof. Erkki Oja, Dr. Pasi Koikkalainen and Dr. Jouko Lampinen. During the time I visited them at the Information Technology Laboratory at Lappeenranta University of Technology I got many important impulses for my later research work. Likewise I would like to thank Prof. Ed Davis for kindly inviting me to visit North Carolina State University (NCSU) to work on the subject of nonconforming computers. It was a pleasure to cooperate with him on Paper F.

I also acknowledge the financial support to REMAP from STU and NUTEK (Swedish National Board for Industrial and Technical Development). The stay at NCSU was supported by the Luleå University of Technology Board of Graduate Studies and Research.

Tack! Tack!

REFERENCES

[1] Arbib, M. A., *Metaphorical Brain 2: An Introduction to Schema Theory and Neural Networks*, Wiley-Interscience, 1989.

[2] Arbib, M. A., "Schemas and neural network for sixth generation computing," *Journal of Parallel and Distributed Computing*, vol. 6, no. 2, pp. 185-216, 1989.

[3] Bengtsson, L., A. Linde, T. Nordström, B. Svensson, M. Taveniku and A. Åhlander, "Design and implementation of the REMAP ^{3} software reconfigurable SIMD parallel computer ," in *Fourth Swedish Workshop on Computer Systems Architecture*, Linköping, Sweden, 1992.

[4] Bengtsson, L., A. Linde, B. Svensson, M. Taveniku and A. Åhlander, "The REMAP massively parallel computer platform for neural computations," in *Third International Conference on Microelectronics for Neural Networks (MicroNeuro '93)*, Edinbur gh, Scotland, UK, pp. 47-62, 1993.

[5] Blank, T. and J. R. Nickolls, "A grimm collection of MIMD fairy tails," in *Frontiers of Massively Parallel Computation (Frontiers ´92)*, H. J. Siegel Ed., McLean, Virginia, USA, pp. 448-457, 1992.

[6] Blevins, D. W., E. W. Davis, R. A. Heaton and J. H. Reif, "Blitzen: A highly integrated massively parallel machine," *Journal of Parallel and Distributed Computing*, vol. 8, pp. 150-160, 1990.

[7] Dally, W. J., "Performance analysis of *k*-ary *n*-cube interconnection networks," *IEEE Transactions on Computers*, vol. 39, no. 6, pp. 775-785, 1990.

[8] Feng, T.-Y., "A survey of interconnection networks," *IEEE Computer*, no. 12, pp. 12-27, 1981.

[9] Fernström, C., I. Kruzela and B. Svensson, *LUCAS Associative Array Processor - Design, Programming and Application Studies*, vol. 216 of *Lecture Notes in Computer Science*, Berlin: Springer Verlag, 1986.

[10] Flynn, M. J., "Some computer or ganizations and their ef fectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948-60, 1972.

[11] Fox, G. C., "What have we learnt from using real parallel machines to solve real problems?," Caltech report C^3P-522, 1989.

[12] Hockney, R. W. and C. R. Jesshope, *Parallel Computer 2*, Adam Hilger imprint by IOP Publishing Ltd., 1988.

[13] Holler, M. A., "VLSI implementations of neural computation models: a review," in *Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S. Touretzky Eds. Denver, CO, USA, pp. 993-1000, 1990.

[14] Larsson-Edefors, P., "A 470-MHz CMOS true single phase clocked bit-serial arithmetic unit," *IEEE Transactions on Circuit and Systems, I: Fundamental Theory and Applications*, vol. 41, no. 4, pp. 337-341, 1994.

[15] Lawson, H. W., "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, 1992.

[16] Lawson, H. W., *Parallel Processing in Industrial Real-Time Applications*, Englewood Cliffs, NJ, USA: Prentice-Hall, 1992.

[17] Leiserson, "Fat-trees: universal networks for hardware-ef ficient supercomputing," *IEEE Transactions on Computers*, vol. 34, pp. 892-901, 1985.

[18] Nilsson, K., "Time-deterministic communication in a computer system for embedded real-time applications," Licentiate Thesis 179L, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1994.

[19] Nilsson, K., B. Svensson and P.-A. Wiberg, "A modular, massively parallel computer architecture for trainable real-time control systems," *Control Engineering Practice*, vol. 1, no. 4, pp. 655-661, 1993.

[20] Ohlsson, L., "An improved LUCAS architecture for signal processing," Tech. Rep., Dept. of Computer Engineering, University of Lund, 1984.

[21] Taveniku, M. and A. Linde, "A reconfigurable SIMD computer for artificial neural networks," Licentiate Thesis 189L, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1995.

[22] von Neumann, J., *The Computer and the Brain*, New Haven: Yale University Press, 1958.

[23] Wiberg, P.-A., "Change-oriented time-deterministic real-time systems: motivation and implementation," Licentiate Thesis 177L, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1994.

[24] Åhlander, A. and B. Svensson, "Floating point calculations in bit-serial SIMD computers," Research Report CCA-1992, Centre for Computer Architecture, Halmstad University, 1992.

Davis, E. W., T. Nordström and B. Svensson, “Issues and applications driving research in non-conforming massively parallel processors,” in *Proceedings of the New Frontiers, a Workshop of Future Direction of Massively Parallel Processing*, I. D. Scherson Ed., McLean, Virginia, 1992, pp. 68-78.

Issues and Applications Driving Research in Non-Conforming Massively Parallel Processors

Edward W. Davis[+], Tomas Nordström[‡], and Bertil Svensson*

[+] North Carolina State University
Raleigh, North Carolina

[‡] Luleå University of Technology
Luleå, Sweden

* Chalmers University of Technology
Gothenburg, Sweden

Abstract

Concepts such as modularity and heterogeneity are becoming important for a growing number of applications that use massively parallel computer architectures. Application areas which seem to require these concepts appear in real world computing and action oriented systems. In many instances the current offerings of high performance, parallel, general purpose computers are not well suited to these applications since they do not address issues like real-time, time determinism, heterogeneous communication, physical size, power consumption, etc. These issues are important in special systems that can be viewed as non-conforming to general purpose markets. The differences in needs will be explored by looking into two examples of modular and heterogeneous systems: high performance instrumentation systems and action oriented systems. We raise some research issues that need to be resolved in order for modular and heterogeneous systems to be used effectively and efficiently.

Key words:

Massively Parallel, Non-Conforming Computers, Modular, Heterogeneous, Embedded, Real-World Computing, Real-Time, Action Oriented Systems.

1. Introduction

The goal of the Frontiers '92 Workshop on Processor Architectures was to identify problems that could be addressed through research, and whose solutions would promote the availability and use of massively parallel processing. In recent years the meaning of "massively parallel processing" has broadened. In 1986, when the first Frontiers symposium was held, the phrase meant systems with more than 1000 processing elements, and the architectural model was definitely SIMD. Now it rightly includes MIMD architectures, and variations on these two models. It is less right, however, to call systems with 32 processors massively parallel, as has been done in some instances. In this paper our view of architectural models is quite liberal, encompassing heterogeneous computing environments, and we definitely are concerned with systems having thousands of processors.

The workshop produced discussion on a broad range of research issues. We expand on the discussion in selected areas and identify problems and research issues from those areas. The nature of this paper, consistent with the workshop, is not to report research results. It is to identify research that needs to be done. Our interest is not systems targeted for general purpose, high performance computing. It is systems that are in some ways special purpose or application specific. The use of "non-conforming" in the title is meant to indicate systems that differ in substantial ways from the commercial offerings, or that have unique requirements that are not well met by current offerings. We begin by defining terms and concepts.

Two important concepts in this paper are modularity and heterogeneity. By "modular" we mean that a suitable architecture can be achieved by combining a number of building blocks (modules). Each module can be a computer in its own right, and in our context each module could be a homogeneous parallel computer module as well. "Heterogeneous" indicates that these modules can be of different kinds, that is, they can differ in parallelism, control, I/O support, and other aspects of their architecture. By having different kinds of modules it is possible to use a module that fits a certain part of the application very well, and by combining modules we get a very good fit between an application and the architecture for many applications. An abstract view of such a system is given in Figure 1, where there are three modules of different types which communicate with each other and with the external world of peripheral devices, instruments, sensors, actuators, etc.

There are several additional concepts of importance. The notion that a system is "resource adequate" means that computational resources, including I/O, have the necessary power to accomplish the task in an allotted amount of time. "Embedded" systems are just those in which resources are closely coupled to other parts of a system, such as sensors or actuators. Having the capability to configure modular, heterogeneous systems means that we can achieve embedded systems with resource adequacy, which relieves us from many of the real-time resource sharing problems.

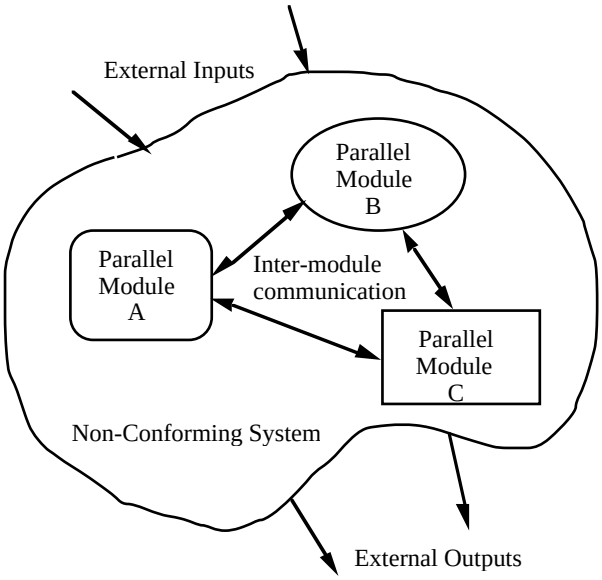

Figure 1. A modular, heterogeneous, parallel system.

By studying two areas where modular and heterogeneous architectures are suggested, specifically instrumentation systems for large experiments and action oriented systems, many important architectural aspects will be found. In addition to the two areas we mention, similar problems and opportunities can be found in applications originating in manufacturing, military, medical, environmental, spaceborne, and other endeavors.

In major experiments, such as the new particle colliders proposed in the U.S. and Europe, data must be acquired from thousands of physically distributed sensors. Data rates are high and real-time processing is needed to select important data and reject the rest. Parallel processing can be used to advantage on the images of particle tracks and energies. However, large packaged parallel machines are not very suitable since processing requirements and the physical environment makes it necessary to modularize and embed processing resources in proximity to the sensors. Other characteristics of this environment, and the research issues it raises, are further discussed in Section 2.

In an action oriented system, sensory, motor, and processing parts, all possibly utilizing neural network principals, are seen as an integrated system capable of interacting with the environment in real-time. Integrated should not mean that there is only one block of computation, instead it should be seen as a number of cooperating smaller blocks. Each block is carrying out different styles of signal processing, e.g. pattern recognition, vector quantization, or error correction. Many blocks are potentially implemented as artificial neural networks (ANN). This modularization corresponds very well to how the brain is organized, where real neurons often can be found to be grouped into larger structures (hundreds of thousands of neurons). In Section 3 we further explore the idea of action oriented systems.

1.1 Demands placed on systems

The intended application areas, and the key concepts, lead to a number of demands on these massively parallel systems. The first demand is *real-time performance*. The implication of this demand is that for each task there must always be enough computational and I/O resources guaranteed. Since hardware is often cheap it is natural to use resource adequacy as a hardware design philosophy. Different tasks require different computing paradigms and system architectures. As a consequence, the final system may be heterogeneous. Therefore, we need to find overall system architectures in which we can still deal with, and guarantee, the real-time demands.

The second demand is *embedded implementations*. Miniaturization and low power consumption are necessary to achieve an embedding of resources. By taking advantage of the advances made in VLSI technology and packaging, e.g. multichip module techniques, the goals of both miniaturization and low power consumption can be met. In many cases embedding also leads to distributed systems, which in turn implies that modularization is required. Communication needs can be critical. Processing is frequently required to be close to sensors and massively parallel I/O becomes an important issue.

The third demand is support for *safety critical functions* and preparedness for *harsh environments*. Fault tolerant processor arrays and communication, and/or fault tolerant computational models are increasingly important for applications where human safety is involved. Alternatively, fault tolerance is needed when applications require computing equipment to operate in places where conditions are harsh or repair is difficult.

The fourth demand is the ability to function in dynamic, *real-world environments*. Adaptability to changing environments, and self-organization in relation to input patterns that have never before been encountered, are necessary functions. An emerging technology in order to achieve such advanced behavior is the application of neural network principles. A way to cope with the complexities involved with advanced systems functioning in natural

environments is to use a multitude of cooperating ANNs, organized in layers and hierarchies. Support for this must then be given in the architecture.

The fifth, and final, demand that we consider here is the need for *new development methods and tools.* Action-oriented systems, as well as other systems where the environment can not be fully modeled, must be developed through interaction. This interaction exists both between the system designer and the system, and between the system and the environment. Developing/training the system on-line, using the real sensors and actuators, must be supported.

1.2 Contrasting system design goals

Unfortunately the problem areas mentioned above, coupled with the system demands, do not fit very well into the current offerings of high performance, parallel, general purpose computers. General purpose computers do not have the need or luxury to address the range of issues of the embedded systems we are investigating. As examples of different design goals and constraints consider:

General Purpose	*versus*	*Embedded Systems*
• Maximum performance		• Resource adequate
• Throughput oriented		• Real-time, time determinism
• Size is of minor concern		• Size is important
• Standard languages like HPF		• Custom programming
• Normal I/O capabilities		• High I/O bandwidth
• Standard data formats		• Data transformations

When the first massively parallel machines were developed each processing element (PE) was very simple, often bit-serial [4, 16, 18]. Many of the organizations that developed such machines now have moved towards highly parallel computers where each PE is much more powerful [13, 19]. This is partly a result of technology advances, and partly a reaction to the market pressure towards high maximum performance and general purpose usage. As a vendor put it at the workshop: "The market is only interested in using 32-bit data".

Certainly the grand challenge problems present strong motivation and incentive to architects and corporations [9]. While all of the problems require very high computational rates, most do not have real-time requirements. The one exception is weather forecasting where response time, although real, is not short. All can be handled using hardware configured as large systems in controlled environments. This has left the field of real-time, embedded, and action-oriented systems without major support. Many of the design goals of a general purpose highly parallel computer do not apply for this class of machines.

This paper suggests that modular, heterogeneous systems will play an important role in the future use of massively parallel processing. Sections 2 and 3 give examples of these systems and describe some unique aspects and environments of their use. Section 4 emphasizes the research issues that must be addressed for success with these special systems that do not conform to current commercial offerings.

2. High performance instrumentation systems

An informative example of massive parallelism in an embedded real-time system occurs in the field of high energy physics (HEP). Particle colliders like the Superconducting Super Collider (SSC) in the U.S., or the Large Hadron Collider (LHC) in Europe produce very large quantities of experimental data in very short time spans. At projected beam collision intervals of 15 ns, one of the sensing instruments for the new colliders will output data at rates in the neighborhood of 10^{13} bytes/s [3]. The problem is to acquire interesting data from the vast quantity produced, and then to find "events" of interest in the data such as particle tracks and peaks of energy. These results are further processed to determine energy, momentum, time duration, and other aspects that represent the physics of the events.

Hierarchical and parallel configurations of computers are used extensively for data acquisition and processing in the instrumentation systems of colliders [8]. Detectors surround the site of collisions in a physically large volume extending approximately 10 meters along the beam axis and 2 meters in diameter. In the terminology of the physicists, the computer structure is in levels of "triggers". For the LHC the first level trigger acts as a filter by identifying regions of interest within the total set of detectors. The second level trigger typically locates particle tracks or peaks of energy. It thus reduces the data but increases the information passed to a third level where the physics of the events is processed. Our own work is at the second level. A massively parallel processing array based on the Blitzen SIMD device [5] is being evaluated for use by CERN in the LHC [7].

2.1 System properties

This instrumentation system displays many of the properties that were described in Section 1. It also provides examples of research issues that must be addressed to realize trigger structures or comparable systems. Note that SSC and LHC are in the early phases of development, with initial experiments expected in the late 1990's, and thus these issues are current and ongoing research issues.

If we consider the requirements of HEP instrumentation, the need for modular and heterogeneous properties becomes apparent. The system is real-time in that collisions occur at definite time intervals and the interesting

data for a collision must be gathered and processed, at least through the second level where it is reduced such that it can be saved by the third level for off-line physics calculations. Collisions occur at 15 ns intervals. It is the responsibility of the first level to determine which collisions are producing potentially interesting data and to pass that reduced amount of data to the second level. A tentative goal for the second level in LHC is to gather and process at a 100 kHz decision frequency. Thus, the real-time processing interval is just 10 μs. To put this in a computer time context, the overhead for sending one message between processors in the Intel Paragon is about 25 μs.

The system is heterogeneous in that different processing technologies and architectures are used at different levels of the hierarchical arrangement. The general structure is shown in Figure 2 [6]. Data rates differ dramatically, as do processing requirements. SIMD parallel processing arrays are being evaluated for the second level, but MIMD is likely for the third level. Analog devices may be used in the first level. Within a level, the processors are homogeneous, but they must communicate with different types of processors in the other levels.

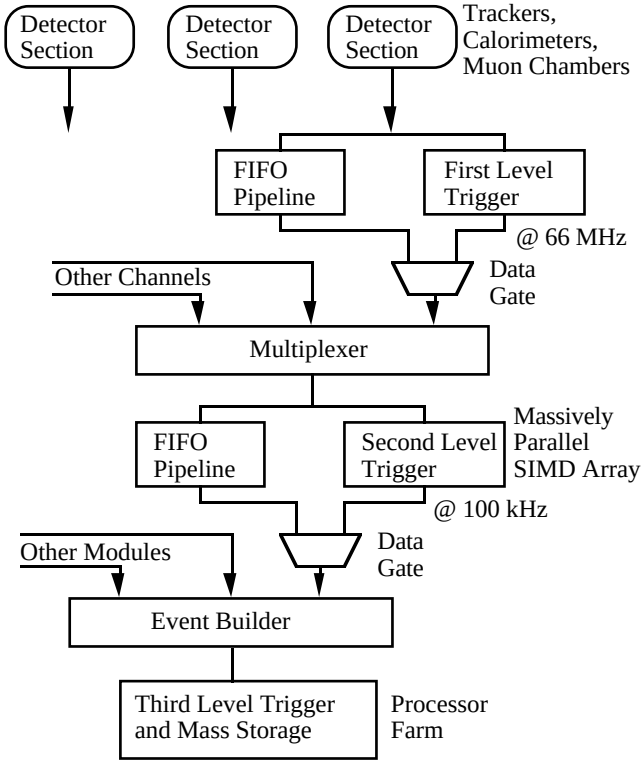

Figure 2. Hierarchical trigger structure with heterogeneous processors.

The system is modular in that processors must be physically near the source of data to accommodate high bandwidth transfers from the sensors and the collision frequency. Since sensors are densely distributed over the 10 m length of the instrument and around the circumference, the three levels in the processing hierarchy are also distributed. There is a tree structure to the hierarchy with the first level being the leaves. Thus modules of SIMD arrays can be used for the second level, and a reduced number of MIMD modules, or possibly just one, for the third level. Essentially, the system is modular since it is necessary to use piecewise coverage of detectors in the collision volume. It is also massively parallel due to the number of processing elements needed to provide the interfaces and processing for those detectors.

A further characteristic of the systems we are studying is that processors are embedded with other electronics and mechanics. This is clearly the case with instrumentation for HEP. It is impractical as well as unworkable to think of running 560,000 wires transferring a cumulative 10^7 megabytes per second, as expected for the transition radiation detector (TRD) [3], from the instrument site to a room with a computer system.

The final characteristic for this example has been introduced in the paragraphs above. There is a high bandwidth I/O requirement. For the TRD example, the major burden is on input with an expected bandwidth of 10^7 MB/s into the first level and $7.7*10^5$ MB/s into the second. In general, some formatting or transformation of data may be necessary. For this example application it is necessary since the instruments produce small outputs that can be represented in a few bits. The bits are gathered into 32-bit words for transmission over HiPPI channels, then must be transformed by a corner turning process for alignment with processing elements.

Figure 3 shows one SIMD module with 1024 PEs as it may be used for the second level trigger. Several such modules are needed for the total system. A region of interest is selected by the first level and delivered via a HiPPI channel to this level. The array of PEs was sized to satisfy I/O and processing rates, and the region is mapped to the array. Results of feature extraction algorithms are passed to the third level. The flow of regions of interest is continuous during an experiment, with a goal of 10 μs for processing each 16 by 240 pixel region.

All of these characteristics are described to emphasize the need for application specific solutions rather than commercially available systems. This became apparent in practice through the process used by CERN to selectively refine the choices for the final system to be used in the LHC. In their process, a progressive set of evaluations is made. They first identified candidate technologies, then specified benchmark tests. Results were presented at a conference held at CERN, in Geneva Switzerland. A candidate commercial massively parallel system had effective decision frequencies that met the desired rate. However, it achieved the good rate only by accumulating a large number of events and processing events in parallel. This pro-

duced long latencies for the earlier events accumulated, followed by a burst of results for all events. The irregular, bursty nature of the processing was not acceptable in the overall design of the instrumentation. The lack of modularity in a fixed commercial system was a detriment in this case. Other researchers using systems with more modularity proposed adequate resources, still using a high degree of parallelism, that more closely matched the problem size. A single event could be processed using parallelism, but events were not accumulated into parallel data sets. Results were produced at the uniform time interval of input data arrival for the events.

2.2 Design problems

Application specific systems like the second level trigger described above frequently require the solution of many interesting problems. There are research issues and research approaches to the problem solutions. We briefly identify some issues that relate to this example, then defer to Section 4 for the main discussion of research direction for these non-conforming massively parallel computers.

An interesting high level design problem is mapping the application, expressed as algorithms and data properties, into an architecture. The problem exists from two points of view: selecting the processing resources and configuring those resources. The richness of devices and architectural arrangements of those devices into solutions provides many parameters that can be exploited during a design phase. Tools and techniques to assist in the mapping could be very useful.

Algorithms and data rates for high energy physics strongly imply a heterogeneous, hierarchical structure. Research is needed in methods for partitioning tasks and communicating through the hierarchy. Fault tolerance and reliability issues must be addressed since experiments are expensive and the system is complex.

Real-time computing in HEP has the constraint of very short response or computation times. Providing real-time for a problem with response times in the microseconds is different from that for one with milliseconds of time. Fortunately, the problem does not require response to a wide set of irregular inputs driven by interrupts. It is real-time from the point of view of needing to complete a fixed routine and perform related I/O within a very short time span. This is well-suited to the resource adequacy notion, assuming sufficiently adequate devices are available.

In the next section we give a second example of applications where standard parallel processing systems are not suitable. It contrasts in several ways to the instrumentation example above, but it also has similarities and presents several of the same research needs.

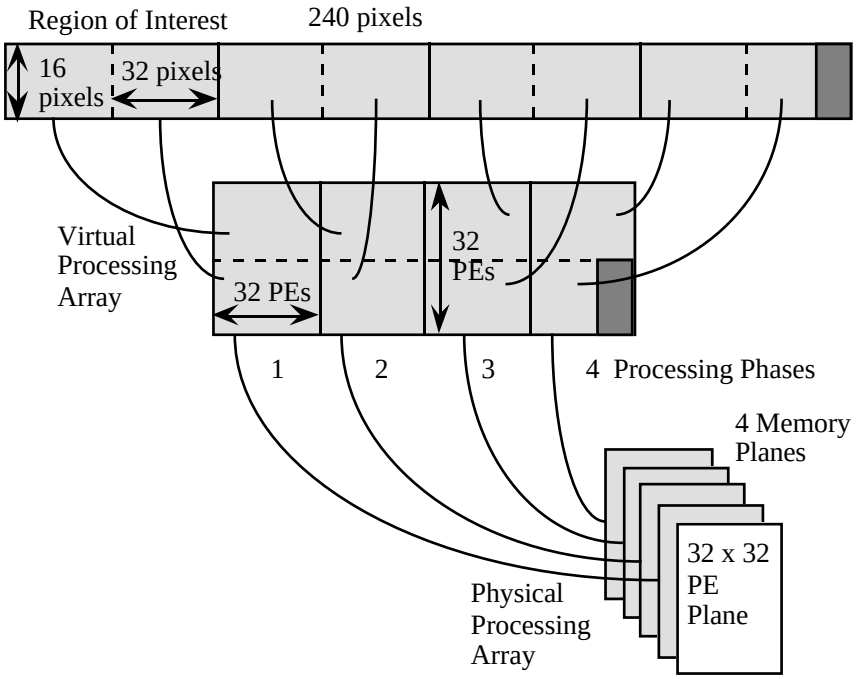

Figure 3. Mapping detector data to a processing array in the second level trigger.

3. Action oriented systems

An action oriented system (AOS) is used as the second example of an application area where modular and heterogenous computers will be needed. Here these aspects are combined with embeddedness, real-time responses, high I/O and internal communication bandwidth, etc., which generates a need for non-conforming massively parallel processing systems.

The concept of action oriented systems (or computing) has been developed by Michael Arbib for many years and is often called "sixth generation computers" by Arbib himself [1]. That particular term has become more appropriate since the introduction of the MITI real world computing (RWC) program in Japan [10], which replaces the much debated fifth generation project. This is because many of the goals of the RWC program are focused on expanding the knowledge in the field of action oriented systems!

Many of the ideas of action oriented systems are drawn from the organization and function of the human brain. The key-concepts of action oriented computing are:

- *Cooperative computing*

Using the brain as a model, we find a number of cooperating areas instead of a large homogeneous information processing facility. Each area is, of course, highly parallel and can for now be approximated as homogeneous in structure. This makes it very natural to suggest a heterogeneous modular computer for simulating action oriented systems. As each module communicates with many other modules in a massively parallel way, this structure puts strong demands on intermodule communication.

- *Perceptual robotics*

An AOS generates its knowledge about the surrounding world by exploration. The system interacts with the environment through a process of Perception -> Decision -> Action. The perception can be sensors for images, speech, tactile information, etc. As these sensors are massive, inexact, and many times incomplete, the system must be highly parallel, robust and fault tolerant. And as real-world information is volatile, the system must work in real-time.

- *Learning*

In contrast to the computers of today which need exact instructions (rules or programs) to function, an action oriented system will base much of its actions on learning. The system should be able to self-organize the information it gains from exploration of the world, and integrate information from many different sources to create an internal model of the world. From past experiences incorporated in the internal model, it should be able to make decisions on what actions are appropriate. Much of this learning will be implemented as artificial neural networks, but certainly any universal or domain knowledge can and should be incorporated in advance (like the laws of Newton).

In Section 1.1 most of the demands of an AOS were summarized. But besides issues mentioned there, the close interaction with sensors and actuators will need attention. This closeness makes it desirable to have the computations take place in the sensors/actuators in a massively parallel fashion. Another issue is development methods for applications using AOSs. As learning instead of programming is emphasized, the possibility to develop/train the system on-line or in-the-loop (using the real sensors and actuators) seems desirable. Still the system must support ways to handle timing constraints in a natural way. Future development environments for AOS should probably be graphically based, using domain specific symbols (hierarchically), and time attenuations [14, 17]

After looking at one early example of an AOS in Section 3.1, we will discuss suitable architectures for AOSs in Section 3.2. Following that, we find that not only will AOS influence research in non-conforming computers, but also the research in the field of artificial neural networks.

3.1 Application areas for action oriented systems

The areas where action oriented systems first will be introduced are in manufacturing, robotics, autonomous vehicles, and the control field in general. As these areas already are action oriented and modular (but not necessarily ANN based) it is not hard to realize that ANN based AOSs are interesting. Often the problems are complex enough to need the complexity of multiple ANNs.

One recent example of an action oriented system is COLUMBUS [20], an autonomous mobile robot developed at CMU. This robot's single goal is to maximize its information of the initially unknown environment. The project has so far concentrated on the algorithms to be used and not so much on the computer architecture to run the algorithms. COLUMBUS uses a mixture of different algorithms (as could be expected of an AOS). There are two separate ANNs for sensor interpretation and confidence estimations. This information is then used to enhance an exploration map at a higher level. By means of a modified dynamic programming algorithm this map is used to decide on an action. The decision is based on where there are unexplored areas and where there are obstacles.

Although it appears that the implementors of COLUMBUS have not concentrated on the architecture, they have produced a modular and distributed implementation, using several SUN SPARC workstations in parallel. As the processing power is not embedded (on the robot) a transmission of sensor and control information by a radio link to and from the robot is needed. By dropping some sensor information, and modifying the dynamic programming algorithm used for planning, it has been possible to reach close to real-time performance (each action taking from 3 to 12

seconds). Even if the authors indicate that they are satisfied with this performance, we feel that a different kind of architecture could help to speed up parts of the system by addressing the problem of embeddedness, I/O performance, and parallelism used.

3.2 Architectures for action oriented systems

The best architecture for an AOS is still an open research question. An architecture we suggested in [17] views the system as a number of *nodes* that communicate through logical *channels*. The real-time concept is supported by demanding time-determinism for all parts in the system. By time-determinism we mean that it should always be possible to determine the execution or cycle time for each computation and communication. To accomplish time determinism we suggest that *local real-time databases* between the nodes and the channels be introduced. The three main concepts of this architecture are described below:

Nodes can be either an I/O interface or a computational entity, or function as a combination. Each node can differ in functionality, and communicate with other nodes via the logical channels. The computation is cyclic [11, 12] and time-deterministic (no interrupts). We expect many of the nodes to be implemented as SIMD computers.

Channels also need to adhere to time-determinism and at the same time give as high communication bandwidth as possible to the communicating nodes. We expect fiber-optics to be used, together with time division multiplexing. If this type of multiplexing is not enough, frequency multiplexing may be considered in addition.

Local real-time databases are needed to store the shared data from other nodes and are updated cyclically from the channels. The data stored in the database reflects the best available information for its node at a certain time.

Figure 4 shows an implementation of this architectural concept. Four Operating Nodes, some incorporating massively parallel I/O and each with a processor array (PE array) connected to a local real-time database (LRTDB), are shown. The Operating Nodes are cyclically controlled by control units (CU). Channels between the nodes are established using time and/or frequency multiplexing on the shared fiber-optic medium.

The figure also shows a Development Node which is connected both to the network of operating Nodes and to a Local Area Network (LAN) of workstations (WS) running the development system. The Development Node may be a PE array (as shown) but may also be another type of computer, but with the same interface to the shared medium. The LAN can be removed without affecting the running system.

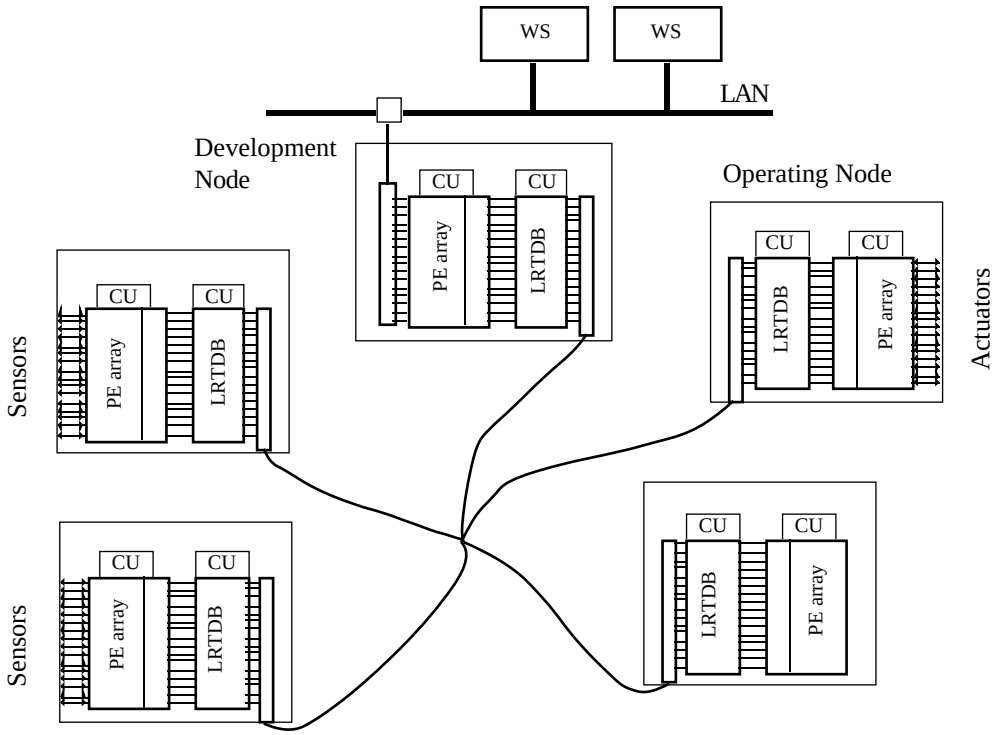

Figure 4. An architecture implementing a modular, heterogenous, parallel system for action oriented computing.

A more detailed description of this architecture concept can be found in [14, 17]. To test these ideas and explore possibilities in the design of the nodes, an experimental system has been built using field-programmable logic devices. Experience from this project, called REMAP [2], will lead to a VLSI design of modules that can be the base for building non-conforming computers, especially in the areas of AOSs. The architecture is intended to be open for the emerging technologies like analog ANN VLSI chips that can do much of the computations close the sensors at extremely high speed.

3.3 AOS Influence on ANN research

During the last ten years a formal explosion of artificial neural network research has lead to a number of different models. Most of the models are naturally parallel and can easily be implemented on highly parallel computers. In a study it has been found that for most ANNs a highly parallel SIMD computer with simple communication (broadcast) is enough [15]. But when it comes to a number of cooperating ANN modules relatively few experiments have been done, and there is no hardware around with the capacity to do real-time simulation of multi-ANN systems big enough to be interesting. Many aspects of multi-ANN are unclear at this time, leading to a need for flexibility in the systems that implement them, to cope with changes in models etc.

Some aspects of ANNs in AOS that need to be addressed are influenced by the real-time aspects. New ANN models need to be developed since real-time creates a need for models that can learn continuously, which is not the case for one of the most popular ANN models (back-propagation learning). And as learning methods using relaxation or structural adaptation are not time-deterministic, it will be hard to use such models for real-time AOS as well. Real-time also means that some types of parallelism in the ANN models [15] can not be used. That is, the training example and training session parallelism are batch oriented and simply are not viable for continuous learning. (We should use the parallelism in weights and nodes instead). Other aspects of ANNs in AOS that need more research are those of planning and creating useful internal world models.

4. Research issues

Very early in our workshop session it became apparent that we could not discuss processor architecture in isolation from the other three workshop topics. They are interrelated. Thus some of the topics below stray from narrow processor issues. Even so, the broader topics all have an impact on processors and must be considered in arriving at processor architecture decisions. Our topics do not include VLSI design issues, even though those also have an impact on processor architecture.

In previous sections we have given examples of modular and heterogenous computers. In each example, general purpose highly parallel computers were not capable of solving the problems within certain performance constraints. Instead we found that a special purpose heterogenous architecture using homogeneous modules was necessary to generate more effective solutions. But before such special systems can become readily available as solutions for a broad class of problems, there are a number of research issues that have to be addressed. Our discussion is representative rather than exhaustive. We highlight some specific issues at the system and processor levels, and then indicate other areas with open research issues.

4.1 Configurable systems

Systems issues are the global concerns that cannot be resolved by considering one PE in isolation, yet they affect each PE's architecture. The systems of interest in this paper are by definition different from those which can be produced in quantity in a commercial manufacturing setting. They are special in some ways and have unique properties. The challenge is to provide the tools, techniques, and methodologies that can lower the cost and improve the quality of systems configured for special applications. From the starting point of a problem specification or an algorithm, coupled with acceptance criteria, how does one arrive at a good system that satisfies the criteria.

An important aspect is that, in the kind of systems we discuss, the modules need not be as much general purpose as processor arrays for "traditional" massively parallel computers. We accept that some modules very strictly follow the SIMD paradigm to be very efficient on some types of computations (and worse on others). For more irregular computations, other paradigms (SPMD, MIMD,...) are used in the modules. That is, we accept heterogeneity in the system. Assuming that our problem is sufficiently large that parallelism becomes advantageous, several more specific issues can be identified:

- How is the overall problem partitioned into modular parts?
- Which style of parallelism (SIMD, MIMD, combination) is appropriate? Can we know from problem parameters?
- Which devices, preferably commercially available, provide the best fit with the problem?
- How do we size a system to provide resource adequacy?
- Does an architecture scale from prototype to full size?
- Can we logically rearrange resources due to lasting changes in the environment or the task to achieve more generality?
- How can we specify benchmarks such that they provide the desired insight to design alternatives?
- Can custom configurations be developed quickly for evaluation?
- How is the evaluation process controlled, given the richness of possibilities with parallel architectures?

4.2 Processor architectures

Individual processor issues are more directly related to the processing power and flexibility of each processor. An architectural decision affecting a processor will co-influence system decisions such as array size. Thus, to do a good processor design, one can not look only at maximum MIPS or FLOPS, but must look at the whole system's performance and the environment in which the system is supposed to operate. Several specific issues are:

- What functional capability is given at each processor?
- What application oriented features for artificial neural networks, associative processing, signal processing, etc. are needed?
- What local control features should be implemented for SIMD processors?
- How is massively parallel I/O incorporated?
- How is memory capacity and access balanced with processing resources?
- What processing granularity is best, from bit-serial processors to full 64-bit widths?
- What interprocessor communication granularity is best?
- In order to achieve maximum performance per watt, what trade-off should be made between clock speed and number of PEs per chip?

4.3 Communications and I/O

As seen in Figure 1, there are communication paths between the system and the external environment, and between modules of the system. If individual modules are parallel processors there is also intra-module communication. Since interconnection networks is the subject area of one of the other workshops, we mention here only the aspects that seem especially important. In general, the problems are all concerned with high bandwidth movement of data between different types of modules with different I/O mechanisms and structures. In many cases, processing is required to be close to sensors or actuators, which may deal with analog signals. Efficient methods for data format conversion and corner turning are needed. Interfaces for HiPPI and other high speed channels must be developed for continuous modes of operation.

4.4 Fault Tolerance

Systems to be used in safety critical and/or harsh environments need to be fault tolerant. Many new possibilities of fault tolerance have emerged with ANN models. This, together with a multi-modular structure of ANN, raises many research issues on how to combine the fault tolerance in ANN structures with fault tolerance in the hardware structure.

- How do we retain the inherent fault-tolerance characteristics of ANNs when we map them onto a processor array?
- What are the methods to distribute fault detection and correction over a large number of modules?
- How can one reason about correctness in time as well as values, in an action oriented framework?

Redundancy and reconfigurability can be used to increase chip yield as well as provide reliability. This becomes increasingly important as VLSI technology allows more processors per chip and die size increases. Research is needed to investigate reconfiguration methods for various interconnection schemes.

4.5 Software development paradigms

As mentioned in Section 3 new development paradigms are needed for non-conforming massively parallel computers. This is especially apparent for AOSs where an on-line or in-the-loop application development method is needed. Some key-concepts for new paradigms will be: graphical interface, data visualization, data parallelism, incremental development on a running system, and software component reuse. All of these concepts need more research, especially concerning their use in systems such as we are describing.

4.6 Artificial neural networks

There are still many challenges for ANN researchers before large modular ANN (AOS) can be built and its function understood. Some of the research issues are stated in Section 3, and many more can be found in the article by Kahaner describing the MITI's real world computing program [10], many of the research issues regarding multi ANNs are listed. For example, the aspects of how the AOS should handle information, its representations, storing and recalling information, integration of multiple information sources, etc. Other research aspects are concerned with the ways learning and self-organization are best carried out. The results of this research have great influence on both system and module architecture of the computing platform

5. Final comments

Identifying research topics has the problem of knowing where to start, and then, where to stop. Given the general area of processor architectures for massively parallel systems, we have chosen to emphasize systems which are unique or special in some way. We refer to these systems as non-conforming since they differ from commercial offerings. The advantage is that they provide a rich environment, one with many degrees of freedom, and perhaps some difficult constraints, for architectural and related research. Two quite different system examples were presented. They were intended to show that massive parallelism can be used in various application areas and to show the need for further research to achieve the desired end results.

Acknowledgments

We would like to acknowledge the support of several organizations for enabling our participation in the Frontiers '92 Workshop and the preparation of this paper. Davis acknowledges support of the U.S. Environmental Protection Agency through Cooperative Agreement CR 820636-01-0, and the North Carolina Supercomputing Center for support during a leave from North Carolina State University (NCSU) and for the provision of computing resources. Nordström acknowledges support of The Board of Postgraduate Studies and Research at Luleå University of Technology through a visiting scholarship to NCSU (920616 – FNTNr 215/91). Svensson acknowledges the support for the REMAP project from The Swedish National Board for Industrial and Technical Development (NUTEK) under contracts No. 900-1583 and 900-1585. And finally we all acknowledge the bright ideas, stimulating discussions, and hard work of the Blitzen groups at NCSU and the University of Padova in Italy, and the whole REMAP group at Halmstad University, Luleå University of Technology, and Chalmers University of Technology.

References

[1] Arbib, M. A. "Schemas and neural network for sixth generation computing." *Journal of Parallel and Distributed Computing*. Vol. 6(2): pp. 185-216, 1989.

[2] Bengtsson, L., A. Linde, T. Nordström, B. Svensson, M. Taveniku and A. Åhlander. "Design and implementation of the REMAP^3 software reconfigurable SIMD parallel computer." *Fourth Swedish Workshop on Computer Systems Architecture*, Linköping, Sweden, 1992.

[3] Bialas, P., J. Chwastowski, P. Malecki and A. Sobala. "Benchmarking with data from the transition radiation detector." (CERN/EAST note 91-11), CERN, Geneva, Switzerland, 1991.

[4] Blank, T. "The MasPar MP-1 Architecture." *Proceedings of COMPCON Spring 90*, pp. 20-24, San Francisco, CA, 1990.

[5] Blevins, D. W., E. W. Davis, R. A. Heaton and J. H. Reif. "Blitzen: A highly integrated massively parallel machine." *Journal of Parallel and Distributed Computing*. Vol. 8: pp. 150-160, 1990.

[6] Bock, R. K. et al. "Embedded Architectures for Second level Triggering in LHC experiments (EAST)." (CERN/DRDC/90-56), CERN, Geneva, Switzerland, 1990.

[7] Centro, C. S., E. W. Davis, P. Ni, D. Pascoli and E. Siliotto. "Results of second level trigger algorithms using the Blitzen parallel machine." *Proceedings of the 1992 Conference on Computing in High Energy Physics*, C. Verkerk and W. Wojcik ed., Annecy, France, 1992.

[8] CHEP 92. *Proceedings of the 1992 Conference on Computing in High Energy Physics*, C. Verkerk and W. Wojcik ed., Annecy, France, 1992.

[9] GC. "Grand challenges: high performance computing and comunnications". *A report by the committee on physical, mathematical, and engineering sciences*. National Science Foundation. Washington, DC. 1992.

[10] Kahaner, D. "Special report: MITI's real world computing program." *IEEE Micro*. (August): pp. 70-79, 1992.

[11] Lawson, H. W. "Cy-Clone: an approach to the engineering of resource adequate cyclic real-time systems." *The Journal of Real-Time Systems*. Vol. 4(1): pp. 55-83, 1992.

[12] Lawson, H. W. and B. Svensson. "An architecture for time-critical distributed/parallel processing." *Euromicro Workshop on Parallel and Distributed Computing*, Gran Canaria, Spain, 1992.

[13] MP2. "Second-generation MPP system boosts speed fivefold: MasPar array 32-bit CPUs." *Electronic Engineering Times*. (October 5): pp. 14, 1992.

[14] Nilsson, K., B. Svensson and P.-A. Wiberg. "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, 1992.

[15] Nordström, T. and B. Svensson. "Using and designing massively parallel computers for artificial neural networks." *Journal of Parallel and Distributed Computing*. Vol. 14(3): pp. 260-285, 1992.

[16] Potter, J. L. *The Massively Parallel Processor*. MIT Press. Cambridge, Mass. 1985.

[17] Svensson, B., T. Nordström, K. Nilsson and P.-A. Wiberg. "Towards modular, massively parallel neural computer." *Swedish National Conference on Connectionism, SNCC'92*, Skövde, Sweden, 1992.

[18] Thinking Machines Corporation. "Connection Machine, Model CM-2 technical summary." (Version 5.1), T M C Cambridge, Massachusetts, 1989.

[19] Thinking Machines Corporation. "The Connection Machine CM-5 technical summary.", T M C Cambridge, Massachusetts, 1991.

[20] Thrun, S. B. "Exploration and model building in mobile robot domains." *Proceedings of the IEEE International Conference on Neural Networks*, San Francisco, CA, 1993.

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 14, 260–285 (1992)

Using and Designing Massively Parallel Computers for Artificial Neural Networks

TOMAS NORDSTRÖM AND BERTIL SVENSSON*

Division of Computer Science and Engineering, Department of Systems Engineering, Luleå University of Technology, S-95187 Luleå, Sweden

During the past 10 years the fields of artificial neural networks (ANNs) and massively parallel computing have been evolving rapidly. In this paper we study the attempts to make ANN algorithms run on massively parallel computers as well as designs of new parallel systems tuned for ANN computing. Following a brief survey of the most commonly used models, the different dimensions of parallelism in ANN computing are identified, and the possibilities for mapping onto the structures of different parallel architectures are analyzed. Different classes of parallel architectures used or designed for ANN are identified. Reported implementations are reviewed and discussed. It is concluded that the regularity of ANN computations suits SIMD architectures perfectly and that broadcast or ring communication can be very efficiently utilized. Bit-serial processing is very interesting for ANN, but hardware support for multiplication should be included. Future artificial neural systems for real-time applications will require flexible processing modules that can be put together to form MIMSIMD systems. © 1992 Academic Press, Inc.

1.0. INTRODUCTION

This paper is intended to provide a survey of the use and design of massively parallel computers for artificial neural networks (ANNs) and to draw conclusions based on reported implementations and studies. The simple control structure that characterizes massively parallel computers can be SIMD (Single Instruction stream, Multiple Data streams) or a highly restricted form of MIMD (Multiple Instruction streams, Multiple Data streams) that we call SCMD (Same Code for Multiple Data streams).

We try to identify the architectural properties that are important for simulation of ANNs. We also emphasize the importance of the mapping between algorithms and architecture. ANN computations are communication intensive, a fact which may put strong demands on the communication facilities of the architecture. Moreover, the requirements vary with the ANN model used and the mapping between the algorithm and the architecture.

The paper is organized into three parts: The first part (Sections 1 through 7) is ANN-oriented. It concentrates on ANN models and those characteristics of the models that are of interest when considering parallel implementation. In this part we first go through the basics of artificial neural networks and ANN algorithms. We then discuss some general computational topics that are relevant for the implementation of any ANN model, such as the precision of the calculations and the opportunities for parallel execution. We conclude the ANN-oriented part with a discussion of different measurements of speed for ANN computations.

The second part (Sections 8 and 9) is architecture-oriented. Here we define different classes of parallel computer architectures and give a review of the types of ANN algorithms that have been implemented on computers of these classes.

In the final part of the paper (Section 10) we analyze what experiences can be drawn from the reported implementations and try to determine what requirements will be placed on massively parallel computers for ANN simulation in the future—in batch processing, in real-time applications, and in action-oriented systems. In real-time applications, the speed of the input data flow and the requirements for output data are set by the environment. In action-oriented systems, sensory, motor, and processing parts, all possibly utilizing neural network principles, are seen as integrated systems capable of interacting with the environment. These systems are sometimes called "sixth-generation computers" [2, 3].

2.0. THE BASICS OF ARTIFICIAL NEURAL NETWORKS

In this section we describe the basic properties and terminology of biological neurons and networks. We also show some simple models of these biological structures.

It should be noted that ANNs are often far from being good biological models. Instead they may be seen as biologically inspired algorithms. Studying "the real thing" will give perspective on how simple our models are and how complex the brain is.

* Also at Centre for Computer Science, Halmstad University, S-30118 Halmstad, Sweden.

0743-7315/92 $3.00

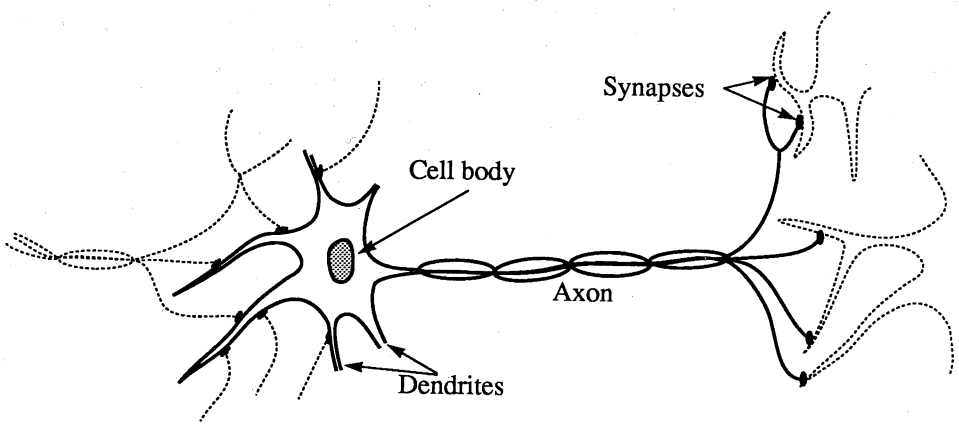

FIG. 1. The principal components of a basic neuron. The input comes to the neuron through synapses on the dendrites. If there are enough stimuli on the inputs there will be an activation (impulse) through the axon which connects to other cells via synapses.

2.1. The Biological Neuron

The basic building block of the brain is the nerve cell (neuron). In humans there are about 10^{12} neurons. Neurons come in many varieties. They are actually all different but can be grouped into at least 50 types of cells.

The principal components of a neuron are shown in Fig. 1. There is a cell body, a number of dendrites (input), and an axon (output). The axon splits and connects to other neurons (or muscles, etc.) The connections function like a sort of chemical resistor and are called synapses. Thus the complexity of the brain is not limited to the vast number of neurons. There is an even larger number of connections between neurons. One estimate is that there are a thousand connections per neuron on average, giving a total of 10^{15} connections in the brain.

Neurons can often be grouped naturally into larger structures (hundreds of thousands of neurons). It has been established that some groups/areas of the brain are organized in a way that reflects the organization of the physical signals stimulating the areas, i.e., topological order. The result is that nearby areas in the brain correspond to nearby areas in signal space. This order is accomplished even when the fibers that are transporting the signals do not exhibit any apparent order. The order seems also to be achieved without any guidance as to what is right or wrong. The resulting maps are therefore often called self-organizing maps. Examples are visual and somatosensory cortex. Each of these structures often connects to other structures at a higher level.

2.1.1. Adaptation and Learning

The brain would not be as interesting, nor as useful, without its ability to adapt and to learn new things. There are basically two ways in which adaptation takes place, by changing the structure and by changing the synapses. The first has the nature of long-term adaptation and often takes place only in the first part of an animal's life. The second, changes of synapses, is a more continuous process that happens throughout the animal's entire lifetime.

2.1.2. Information Processing

The information processing in a neuron is done as a summation or integration of information fed into it. The information is represented as brief events called nerve impulses.† The interval or frequency conveys the information. According to Hubel [50] the impulse rates may vary from one event every few seconds or even slower to about 1000 events per second at the extreme upper limit. The normal upper limit is often cited to be 100 to 200 impulses per second. The "speed" of the impulses along the axon is around 0.1 to 10 m/s. The length of an axon varies from less than a millimeter to more than a meter.

2.2. The Artificial Neuron

The first and very simple model, however much used, is the model in which information is contained as levels/values corresponding to the impulse frequencies. Then the integration of pulses is done as a summation. The synapses are represented as weights, w_j, multiplied by inputs i_j. To make the model more powerful, a nonlinear function, f, is applied to the sum, and the result, $o = f(\Sigma w_j i_j)$, is sent to the neurons connected to it (Fig. 2).

As with their biological counterparts the artificial neurons are not very interesting by themselves. A large number of artificial neurons are necessary for interesting computations. By changing the structure of the connections and adaptation rules it is possible to radically change the type of computations made by the network. Some of the models used are described in Section 3.0.

† This is not true for all neurons. There are, for example, neurons in the retina which have "graded" response. See e.g. [50] or [100] for more on this topic.

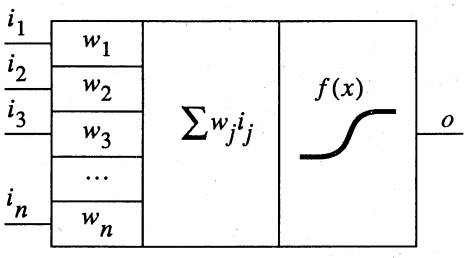

FIG. 2. The simplest model of a neuron. It can be seen as a model of Fig. 1. The output has the form $o = f(\sum w_j i_j)$.

2.3. Layered Models

In many models there are layers of neurons; see Fig. 3. There has been some confusion about how to count the number of layers. One method is to count the node layers including the input layer, and another method is to count weight layers (or node layers excluding the input layer). In this paper we use the word "node" or "weight" in front of the word "layer" when it is needed to avoid confusion. When we count layers we use weight layers, since this is the most relevant method when considering the computational effort. This method of counting implies that one (weight) layer is the smallest network possible. This single-layer network corresponds to the concept of perceptrons [109]. Node layers which have no connection to input or output are called hidden layers; e.g., in Fig. 3 there are two hidden layers.

A compact way of giving the size of a multilayer network is to present the sizes of the node layers with an "×" in between. For example, 203 × 60 × 26 states that the input node layer has 203 nodes, the hidden node layer has 60 nodes, and the output node layer has 26 nodes. Between each layer a fully connected weight layer is assumed. Thus, we consider this a two-layer network.

3.0. SOME OF THE MOST COMMONLY USED ANN ALGORITHMS

During the past 10 years the artificial neural networks area has developed into a rich field of research. Many new models or algorithms have been suggested. Not all these models have been implemented on parallel computers. This is not to say that some of them are not suitable for parallel execution. On the contrary, a common characteristic of all neural network algorithms is that they are parallel in nature. For the purposes of this paper, however, we review the most common ANN algorithms only, in order to be able to discuss their implementation on parallel computers.

The models are characterized by their network topology, node characteristics, and training rules [76]. We describe some frequently used and discussed models.

1. *Multilayer feedforward networks* with supervised learning by *error back-propagation* (BP), also called *generalized delta rule* [110]. The feedforward back-propagation model is used as a pattern classifier or feature detector, meaning that it can recognize and separate different features or patterns presented to its inputs.

2. *Feedback networks* (also referred to as *recurrent networks*). Different variations in node topology and node characteristics have been proposed: symmetric connectivity and stochastic nodes: *Boltzmann machines* [41, 42, 50]; symmetric connectivity and deterministic nodes: *Hopfield nets* [47, 48, 49, 95] and *mean field theory* [95, 96]; and nonsymmetric connectivity and deterministic nodes: *recurrent back-propagation* (RBP) [1, 99]. The feedback models can be used as hetero- or autoassociative memories, but also for solving optimization problems. Using an ANN as an autoassociative memory means that whenever a portion or a distorted version of a pattern is presented, the remainder of the pattern is filled in or the pattern is corrected.

3. *Self-organizing maps* (SOM), also called *self-organizing feature maps* (SOFM) or *topological feature maps* (TFM), developed by Kohonen [62, 63]. This is one of the more frequently used models with unsupervised learning. SOM, with its learning vector quantization variations (LVQ1-3), is used for vector quantization, clustering, feature extraction, or principal component analysis [63].

4. *Sparse distributed memory* (SDM) suggested by Kanerva [58], who argues that it is biologically plausible. The SDM model has been used, for example, in pattern matching and temporal sequence encoding [57]. Rogers [107] has applied SDM to statistical predictions, and also identified SDM as an ideal ANN for massively parallel computer implementation [106].

3.1. Feedforward Networks: Back-Propagation Learning

A feedforward net with three weight layers is shown in Fig. 3. The network topology is such that each node (neu-

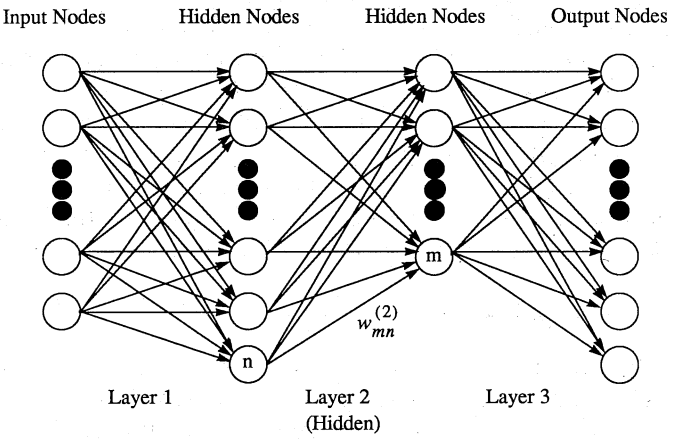

FIG. 3. A three-layer feedforward network.

ron) in a layer receives input from every node of the previous layer. As in most models each node computes a weighted sum of all its inputs. Then it applies a nonlinear activation function to the sum, resulting in an activation value—or output—of the neuron. A sigmoid function, with a smooth threshold-like curve (see Section 4.3), is the most frequently used activation function in feed-forward networks, but hard limiters are also used.

In the first phase of the algorithm the input to the network is provided and values propagate forward through the network to compute the output vector, O. The output vector of the network is then compared with a target vector, T, which is provided by a teacher, resulting in an error vector, E.

In the second phase the values of the error vector are propagated back through the network. The error signals for hidden units are thereby determined recursively: Error values for node layer l are determined from a weighted sum of the errors of the next node layer, $l + 1$, again using the connection weights—now "backward." The weighted sum is multiplied by the derivative of the activation function to give the error value, δ.

Now, finally, appropriate changes of weights and thresholds can be made. The weight change $\Delta w_{ij}^{(l)}$ in the connection to unit i in layer l from unit j in layer $l - 1$ is proportional to the product of the output value, o_j, in node layer $l - 1$, and the error value, δ_i, in node layer l. The bias (or threshold) value may be seen as the weight from a unit that is always on and can be learned in the same way. The algorithm is summarized in Algorithm 1.

Algorithm 1. Back-Propagation Training Algorithm

1. Apply input $O^{(0)} = I$.
2. Compute output $o_j^{(l)} = f(\text{net}_j^{(l)} + b_j^{(l)})$, where $\text{net}_j^{(l)} = \sum_i w_{ji}^{(l)} o_i^{(l-1)}$ for each layer.
3. Determine error vector $E = T - O$.
4. Propagate error backward.
   If node j is an *output* node then the jth element of the error value vector D is
   $\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)})(t_j^{(l)} - o_j^{(l)}) = o_j^{(l)}(1 - o_j^{(l)})e_j^{(l)}$
   else
   $\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)}) \sum_i \delta_i^{(l+1)} w_{ij}^{(l+1)}$.
   Here we have used the fact that the sigmoid function $f(x) = 1/(1 + e^{-x})$ has the derivative $f' = f(1 - f)$.
5. Adjust weights and thresholds:
   $\Delta w_{ij}^{(l)} = \eta \delta_i^{(l)} o_j^{(l-1)}$,
   $\Delta b_i^{(l)} = \eta \delta_i^{(l)}$.
6. Repeat from 1.

By remembering between iterations and adding a portion of the old change to the weight it is possible to increase the learning rate without introducing oscillations. The new term, suggested by Rumelhart and McClelland [110], is called the momentum term and is computed as $w_{ij}^{(l)}(n + 1) = w_{ij}^{(l)}(n) + \Delta w_{ij}^{(l)}(n) + \alpha \Delta w_{ij}^{(l)}(n - 1)$, where α is chosen empirically between 0 and 1. Many other variations of back-propagation exist and some of them have been studied by Fahlman [23].

3.2. Feedback Networks

A feedback network consists of a single set of N nodes that are completely interconnected; see Fig. 4. All nodes serve as both input and output nodes. Each node computes a weighted sum of all its inputs: $\text{net}_j = \sum_i w_{ji} o_i$. Then it applies a nonlinear activation function (see Section 4.3) to the sum, resulting in an activation value—or output—of the node. This value is treated as input to the network in the next time step. When the net has converged, i.e., when the output no longer changes, the pattern on the output of the nodes is the network response.

This network may reverberate without settling down to a stable output. Sometimes oscillation may be desired, otherwise oscillation must be suppressed.

Training or learning can be done in supervised mode with the delta rule [111] or back-propagation [1], or it can be done unsupervised by a Hebbian rule [111]. It is also used "without" learning, where the weights are fixed at start to a value dependent on the application.

3.3. Self-Organizing Maps

Relying on topological ordered maps and self-organization as important concepts, Kohonen developed the SOM [62, 63] which form mappings from a high-dimensional input space into a low-dimensional output space. These maps have been used in pattern recognition, especially in speech recognition, but also in robotics, automatic control, and data compression. The SOM algorithm proceeds in two steps: (i) the network node whose value is closest to the input vector is identified, and (ii) the nodes belonging to the neighborhood of this node (in the output space) change their values to become closer to

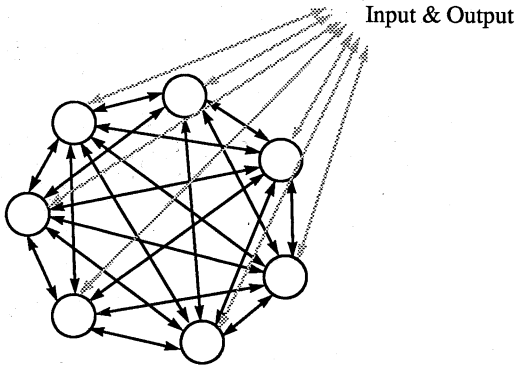

FIG. 4. A seven-node feedback network.

the input. These two steps are repeated with ever-decreasing neighborhood size. The resulting nodes or neurons will develop into specific detectors of different signal patterns.

Described below is the "shortcut version" of the basic SOM algorithm. This version is motivated by the reduction of the computational effort compared to the original one.

Algorithm 2. The SOM Algorithm "Shortcut Version"

1. Find the node (or memory) m_c closest to input x.
$$\|x(t_k) - m_c(t_k)\| = \min_i \|x(t_k) - m_i(t_k)\|.$$
2. Find the neighborhood $N_c(t_k)$.
3. Make the nodes in the neighborhood closer to input.
$$m_i(t_{k+1}) = m_i(t_k) + \alpha(t_k)[x(t_k) - m_i(t_k)] \quad \text{for } i \in N_c(t_k)$$
$$m_i(t_{k+1}) = m_i(t_k) \quad \text{otherwise.}$$
4. Repeat from step 1 with ever-decreasing N_c and α (neighborhood and gain sequence).

With a small change in the update equation (step 3 in Algorithm 2) we can use the same framework for Learning Vector Quantization (LVQ), where the map functions as a clustering or classification algorithm.

3.4. Sparse Distributed Memory

SDM developed by Kanerva [58] may be regarded as a special form of a two-layer feedforward network, but is more often—and more conveniently—described as an associative memory. It is capable of storing and retrieving data at an address referred to as a "reference address." A major difference compared to conventional Random Access Memories (RAMs) is that, instead of having, e.g., 32-bit addresses, SDMs may have 1000-bit addresses. Since it is impossible to have a memory with 2^{1000} locations, SDMs must be sparse. Also, data are stored in counters instead of 1-bit cells as in RAMs

The SDM algorithm has a comparison phase, in which the sparsely distributed locations that are closest to the reference address are identified, and an update (write) or retrieval (read) phase, in which a counter value in each of these locations is used (see Fig. 5).

Algorithm 3. The SDM Algorithm

Training the network (i.e., writing to the memory):

1. The address register is compared to the location addresses and the distances are calculated.
2. The distances are compared to a threshold and those below the threshold are selected.
3. In the selected rows,
   where the data-in register is 1 the counter is incremented,
   where the data-in register is 0 the counter is decremented.

Recall from the network (i.e., reading from the memory):

1. The address register is compared to the location addresses and the distances are calculated.
2. The distances are compared to a threshold and those below the threshold are selected.

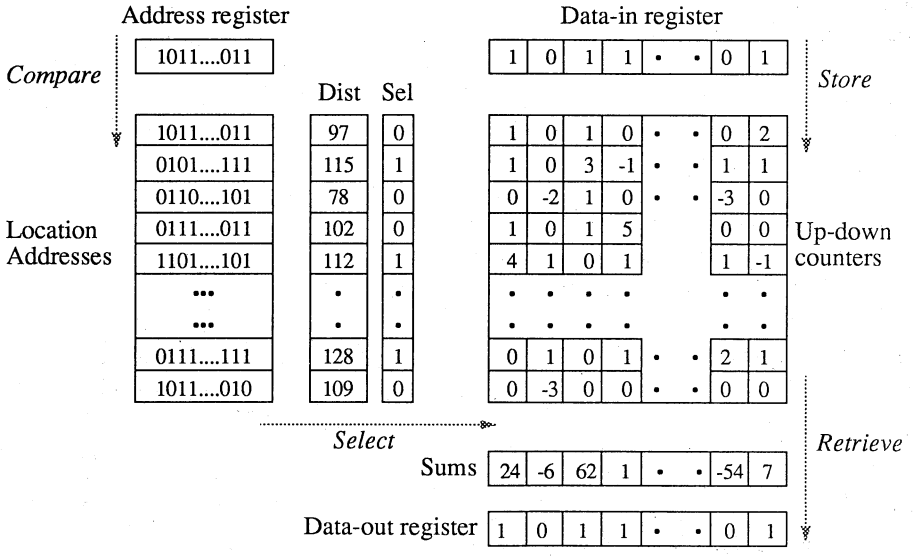

FIG. 5. The organization of a Sparse Distributed Memory as an array of addressable locations. Note that the address as well as the data can be of hundreds of bits in length, and yet there are only a small number (like thousands) of memory locations.

3. The selected rows are added columnwise.
   Where the sum is greater than zero the data-out register is set to one, else it is set to zero.

4.0. COMPUTATIONAL CONSIDERATIONS

The computations involved in neural network simulations show great similarities from one model to another. In this section we discuss some topics that are of general interest and not specific to one single model.

4.1. Basic Computations

For feedforward and feedback network algorithms the basic computation is a matrix-by-vector multiplication, where the matrices contain the connection weights and the vectors contain activation values or error values. Therefore, an architecture for ANN computations should have processing elements with good support for multiply, or even multiply-and-add, operations and a communication structure and memory system suitable for the access and alignment patterns of matrix-by-vector operations.

Assuming N units per layer, the matrix-by-vector multiplication contains N^2 scalar multiplications and N computations of sums of N numbers. The fastest possible way to compute this is to perform all N^2 multiplications in parallel, which requires N^2 processing elements (PEs) and unit time, and then form the sums by using trees of adders. The addition phase requires $N(N-1)$ adders and $O(\log N)$ time.

The above procedure means exploitation of both node and weight parallelism (defined later). For large ANNs this is unrealistic, depending on both the number of PEs required and the communication problems caused. Instead, most of the implementations that have been reported take the approach of basically having as many PEs as the number of neurons in a layer (node parallelism) and storing the connection weights in matrices, one for each layer. The PE with index j has access to row j of the matrix by accessing its own memory. Referring to Algorithm 1, a problem appears in step 4 relative to step 2. While step 2 corresponds to the matrix–vector operation WO, step 4 corresponds to $W^T\delta$. This means that we need to be able to access W^T as efficiently as we can access W. This introduces a mapping problem which we will return to in Section 6. Regardless of the mapping chosen, multiply-and-add is the basic operation in these calculations.

The first step of the SOM algorithm, using an inner-product as distance measure, can also be seen as a matrix-by-vector multiplication, where the matrix is composed of the weight vectors of the nodes and the vector is the training vector. Another distance measure used for the first step is a Euclidean metric which cannot be described as a matrix–vector operation. Still the basic operation in both metrics is multiply-and-add. Discussion on the two different distance measures can be found in [63, 88]. From the resulting vector a maximum or minimum must be found. The efficiency of this operation is strongly dependent on the communication topology, but may also depend on the characteristics of the PEs. In a later section we will demonstrate how bit-serial processors offer specific advantages. After a maximum (or minimum) node is found its neighbors are selected and updated. The selection time will depend on the communication topology, and the update time on the length of the training vectors.

Also in SDM the first step is a matrix-by-vector multiplication. But as both the matrix and the vector are binary valued the multiplications are actually replaced by exclusive-or and the summation by a count of ones. The counters are thereafter compared with a threshold. In all active positions the up-down counters are updated.

Thus, to be efficient for ANN computations computers need to have support for matrix-by-vector multiplications, maximum finding, spreading of activity, count of ones, and comparisons. In some of the implementations that we review these matters have been solved on existing parallel computers, in others new architectures have been devised, targeted at computations of this kind.

4.2. Numerical Precision

In order to optimize the utilization of the computing resources, the numerical precision and dynamic range in, e.g., the multiply-and-add operations should be studied with care.

With optical, analog, or bit-serial computers it is *not* very attractive to use 32- or 64-bit floating-point numbers for weights and activation values. The issue of weight sensitivity becomes important; how sensitive are the networks to weight errors? Unfortunately, one of the most used algorithms, back-propagation, is very sensitive to the precision and number range used [39]. This is due to the existence of large flat areas in the error surface, in which the BP algorithm may have difficulty in determining the direction in which to move the weights in order to reduce the error. To make progress from such areas high numerical precision is needed. If the neural network calculations are run on a computer which has advanced hardware support for floating-point calculations the required accuracy does not raise any problem. On the other hand, for many tasks of integer type, like low-level vision problems, the use of floating-point numbers will lead to a more complex architecture than necessary.

Using ordinary back-propagation with low precision without modifications will lead to instability and the net will often be unable to learn anything. There are modifications to back-propagation which seem to improve the situation somewhat [13, 21, 78, 116], and there are experiments in which different precision is used at different

stages of the algorithm [86]. By using high precision at the beginning of the training and lessening the precision as the network trains, the number of bits needed for weights in the fully trained network can be very low (a few bits). Without too large modifications, 8–16 bits per weight seems to be sufficient for most problems [11, 21, 85, 135]. More exact calculations of the sensitivity to weight errors and the precision needed can be found in [21, 121, 122, 123].

By some authors [39, 135] weights have been found to need a large dynamic range, implying floating-point representation. However, the use of weight saturation, i.e., limiting the weights to a certain limit, may remove the need for floating-point numbers [85].

Low precision is also attractive for ANNs as it makes the algorithms more biologically plausible [131]. An upper limit of the accuracy that the brain needs for its calculation could be estimated to 7–8 bits; i.e., neurons have a "dynamic range" of about 100 levels. The calculations are also fault tolerant. That is, if one neuron fails to fire, the computation is still carried out in the right fashion.

Finally, it should be noted that there is probably a trade-off between using few weights (nodes) with high precision and using many weights (nodes) with low precision.

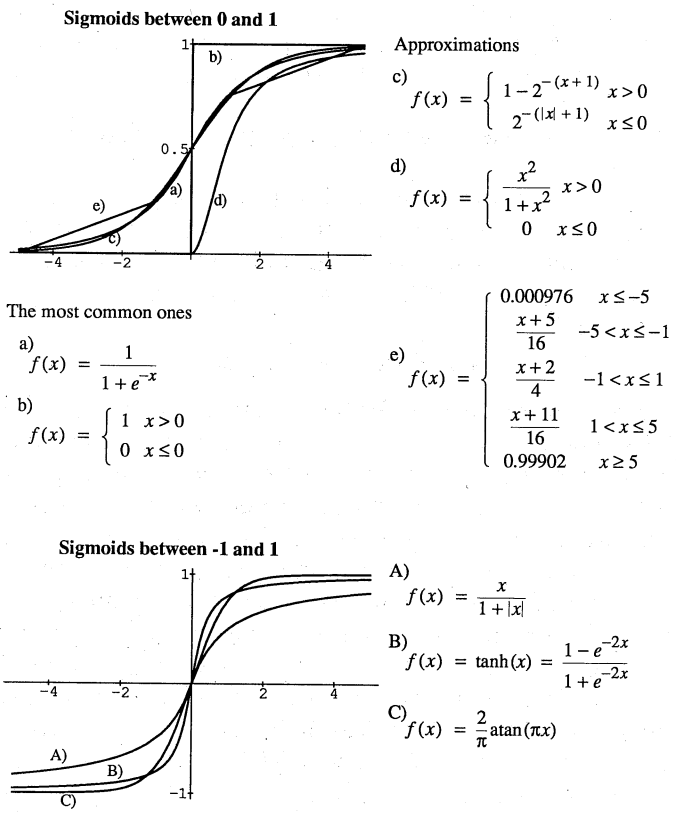

FIG. 6. Some of the used activation functions of sigmoid type.

4.3. Sigmoid

In many of the algorithms, e.g., BP and Hopfield networks, a sigmoid function like $f(x) = 1/(1 + e^{-x})$ needs to be calculated; see Fig. 6. To do this efficiently many implementations use a table lookup instead of direct calculation (typically with 8 bits precision). Others try to approximate the sigmoid with a piecewise linear function [11] like(e) in Fig. 6. Also, an approximation based on power of 2 calculations has been proposed, with digital computers in mind [94], (c) in Fig. 6.

A combination of table lookup and power of 2 calculation was tried in [136] for the GF11 computer, but in the end only table lookup and interpolation were used.

Table lookup on SIMD computers without local address modification seems difficult but is possible by a cyclic rotation and comparison. It takes n steps to do n table lookups using n PEs connected in a ring [132]. Other ways to distribute the lookup table have been discussed by Marchesi *et al.* [80].

In [115] e^x was calculated by means of range reduction techniques. The total number of operations required to calculate the sigmoid was five add/subtracts, one logical, two divisions, two shifts, and three multiplications.

In the backward phase the derivative $f(x)$ is to be calculated. The much used sigmoid $f(x) = 1/(1 + e^{-x})$ has the "nice" derivative given by $f'(x) = f(x)(1 - f(x))$.

Some networks and training situations have turned out to benefit from a sigmoid function between -1 and 1 instead of the usual 0 and 1. Some are given in Fig. 6. The function (A) has the useful property that is does not involve any transcendental functions.

4.4. Data Representation

When real and/or analog inputs to the ANN are used, the data representation must be studied carefully. For the binary code the problem is that Hamming distance (HD) is not a valid measure of similarity; see Table I. The so-called thermometer code, or the simple sum of ones, solves the problem with the HD but is wasteful of code bits, and thus stands in contrast to the requirement of using as few bits as possible.

Penz [93], referring to Willshaw *et al.* [134], has suggested a modification to the thermometer code, called the closeness code. This code has as few active positions as possible; i.e., it is optimally sparse, but still retains the HD as a similarity measure. The number of ones in an N-bit vector is $\log_2 N$.

There is a denser (with respect to code word length) version of a closeness code suggested by Jaeckel [56, 59] which could be called a band-pass code. The name reflects the fact that the thermometer code may be seen as a set of low-pass filters but Jaeckel's suggestion may be seen as a set of band-pass filters. Both a closeness and a

TABLE I
Different Ways to Encode Data: Binary, Thermometer, Closeness, and Band-Pass Codes

Decimal value	Binary		Thermometer		Closeness		Band Pass	
	Code	HD	Code	HD	Code	HD	Code	HD
0	000	0	0000000	0	11100000 00	0	11000	0
1	001	1	1000000	1	01110000 00	2	11100	1
2	010	1	1100000	2	00111000 00	4	01100	2
3	011	2	1110000	3	00011100 00	6	01110	3
4	100	1 !	1111000	4	00001110 00	6	00110	4
5	101	2	1111100	5	00000111 00	6	00111	5

6	110	2	1111110	6	x0000011 10	5/4	00011	4

7	111	3	1111111	7	xx000001 11	4/2	10011	3
							10001	2
							11001	1
							11000	0

Note. All but the binary codes have Hamming distance (HD) as the measure of similarity. Closeness and thermometer codes are wasteful of code bits, but the closeness code has fewer active bits. Band pass has good code density while keeping the HD as the measure of distance.

band-pass code can be extended to a circular code as shown below the dashed lines in Table I. A circular code is useful when coding things like angles.

When coding sets of mutually unrelated items, like letters in an alphabet, it is important *not* to introduce order or similarities that do not exist. Coding a, b, c, ... as 1, 2, 3, ... introduces a similarity between adjacent letters which has no relation to their use in language. Instead, 26 nodes of which only one is active, may be used.

4.5. Bit-Serial Calculations

Many of the massively parallel processors use bit-serial PEs. For the majority of operations, processing times on these computers grow linearly with the data length used. This may be regarded as a serious disadvantage (e.g., when using 32- or 64-bit floating-point numbers), or as an attractive feature (use of low-precision data speeds up the computations accordingly). In any case, bit-serial data paths simplify communication in massively parallel computers.

4.5.1. Multiplication Done Bit-Serially

In simple bit-serial processors the multiplication time grows quadratically with the data length. However, bit-serial multiplication can actually be performed in the time required to read the operands (bit by bit, of course) and store the result. The method, based on the carry-save adder technique, requires as many full adders as the length of one of the operands. Figure 7 shows the design of such a multiplier circuit.

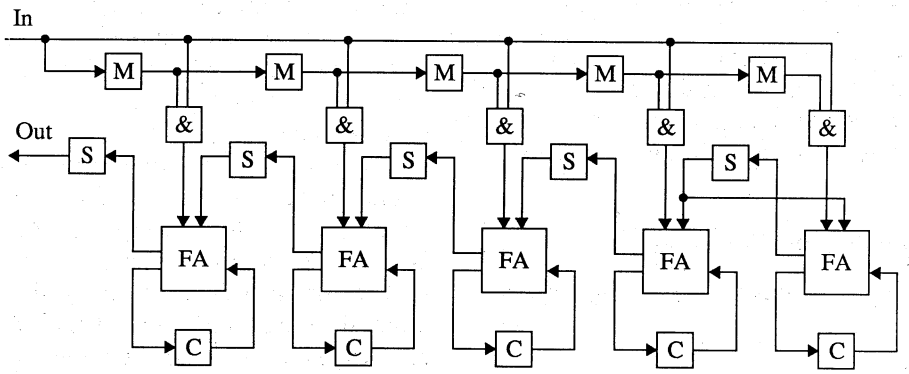

FIG. 7. Design of a two's-complement bit-serial multiplier. It is operated by first shifting in the multiplicand, most significant bit first, into the array of M flip-flops. The bits of the multiplier are then successively applied to the input, least significant bit first. The product bits appear at the output with least significant bit first.

This design was proposed but not implemented in the LUCAS project [27], and will be used in the continued project, REMAP^3 (Reconfigurable, Embedded, Massively Parallel Processor Project). A similar multiplier design has also been proposed for "Centipede," a further development of the AIS-5000 concept [135].

4.5.2. Floating-Point Calculations Done Bit-Serially

Floating-point calculations raise special problems on SIMD computers with bit-serial PEs. Additions and subtractions require the exponents to be equal before the operations are performed on the mantissas. This alignment process requires different PEs to take different actions, and this does not conform with the SIMD principle. The same problem appears in the normalization procedure.

However, these problems may also be solved by a fairly reasonable amount of extra hardware. Åhlander and Svensson [140] propose the addition of stacks in the PEs to hold the mantissas during alignments and normalizations. This arrangement allows floating-point operations to be performed as fast as data can be provided bit-serially from the memory.

4.5.3. Search for Maximum or Minimum Done Bit-Serially

Some operations benefit from the bit-serial working mode and can be implemented very efficiently. Search for maximum or minimum is such an operation. Assuming that one number is stored in the memory of each PE, the search for maximum starts by examining the most significant bit of each value. If anyone has a one, all PEs with a zero are discarded. The search goes on in the next position, and so on, until all bit positions have been treated. The time for this search is independent of the number of values compared; it depends only on the data length (provided that the number of PEs is large enough).

5.0. PARALLELISM IN ANN COMPUTATIONS

For implementation on a parallel computer, parts of the algorithm that can be run in parallel must be identified. Unfolding the computations into the smallest computational primitives reveals the different dimensions of parallelism.

5.1. Unfolding the Computations

A typical ANN algorithm has the following structure:

```
For each training session
  For each training example in the session
    For each layer (going Forward and Backward)
      For all neurons (nodes) in the layer
        For all synapses (weights) of the node
          For all bits of the weight value
```

This shows that there are (at least) six different ways of achieving parallelism:

```
Training session parallelism
  Training example parallelism
    Layer and Forward–Backward parallelism
      Node (neuron) parallelism
        Weight (synapse) parallelism
          Bit parallelism
```

5.2. The Dimensions of Parallelism

Here follows a discussion on each of the different ways of achieving parallelism. Which of the dimensions of parallelism are chosen in any particular implementation will depend on the constraints imposed by the hardware platform and by the constraints of the particular algorithm that is to be implemented.

5.2.1. Training Session Parallelism

Training session parallelism means starting different training sessions on different PEs. Different sessions may have different starting values for the weights, and also different learning rates. Using parallel machines with complex control makes it even possible to train networks of different sizes at the same time.

5.2.2. Training Example Parallelism

The number of training examples used is usually very large, typically much larger than the number of nodes in the network. The parallelism of the training set can be utilized by mapping different training examples to different PEs and letting each PE calculate the outputs for its training example. The weight changes are then summed.

Doing the weight update this way (batch or epoch updating) means that it is not done exactly as in the serial case. Back-propagation is known to perform gradient descent if update of weights takes place after processing of all the training data [110, 119]. Empirically it is found that updating after each pattern will save CPU cycles, at least on a sequential computer.

Training example parallelism is easy to utilize without communication overhead. Thus it gives an almost linear speedup with the number of PEs. However, a corresponding reduction in training time (time to reduce the total error to a specific level) should not be taken for granted. Even if this method gives a more accurate gradient it does not necessarily allow more weight updates to occur. Therefore there is a limit on the amount of actual training time speedup that is achievable using this method of parallelism. Parker [92] has shown that on a

30 × 30 × 10 network and 4156 training examples the optimal batch size was only 18. Beyond that level the refinement of gradient was wasted. This means that the use of extensive number crunching circuitry in order to utilize training example parallelism, although giving high CUPS (Connection Updates Per Second) performance, does not guarantee a corresponding reduction of training time.

Training example parallelism also demands that the PEs have enough memory to store the activation value of all the nodes in a network. With a 256-kbit memory per PE and 32-bit floating-point number representation, only about 8000 nodes can be simulated [119]. On the other hand, any sparsity of the network connections can be fully exploited with this type of parallelism.

5.2.3. Layer and Forward–Backward Parallelism

In a multilayer network the computations may be pipelined; i.e., more than one training pattern is going through the net at the same time. If the model has a backward pass, like BP, it is also possible to "fold" the pipeline back again.

5.2.4. Node (Neuron) Parallelism

The parallel processing performed by many nodes in each layer is perhaps the most obvious form of parallelism in an ANN. Each node computes a weighted sum of all its inputs. This form of parallelism corresponds to viewing the calculations as matrix operations and letting each row of the matrix map onto a processor.

Most of the layered models only send their activation values forward after all nodes have been calculated in a layer. This means that the maximum parallelism is available for the widest node layer (excluding the input layer). If this degree of parallelism is fully utilized, all layers with less nodes cannot fully utilize the computer.

5.2.5. Weight (Synapse) Parallelism

At each input to a neuron the arriving activation value is multiplied by the weight of the specific input. This can be done simultaneously at all inputs to the neuron. The subsequent summation of all the products may also be parallelized using a suitable communication structure.

5.2.6. Bit Parallelism

Utilizing the full degree of bit parallelism (i.e., treating all bits in a data item simultaneously) is often taken for granted. However, giving up this form of parallelism, and treating data bit-serially, increases the possibilities of using some of the other forms.

5.3. The Degrees of Parallelism

The typical degree of parallelism varies widely between the six different kinds, as the table below shows. As an illustration, the degrees of parallelism of the well-known NETtalk application (see section 7.2.1) have been given as well.

Parallelism	Typical range	NETtalk
Training session	$10–10^3$	100
Training example	$10–10^7$	5000
Layer and Forward–Backward	1–6	1
Node (neuron)	$100–10^6$	120
Weight (synapse)	$10–10^5$	203
Bit	1–64	32

The table gives an indication of what dimensions should be utilized in a massively parallel computer. Such a computer is capable of performing at least thousands of elementary operations simultaneously. Hence an ANN implementation that is to utilize the computing resources efficiently must utilize at least one of the following dimensions:

Training session parallelism
Training example parallelism
Node parallelism
Weight parallelism

The use of the two first-mentioned types is of interest only in batch processing situations in order to train a network. In real-time applications where the ANN is interacting with the outside world, training session and training example parallelism are not available. In those cases, node and/or weight parallelism must be chosen, maybe in combination with, e.g., bit and layer parallelism.

6.0. COMMUNICATION

A high degree of connectivity and large data flows are characteristic features of neural computing models. Hence the structure and bandwidth of internal and external (i.e., I/O) communication in the computer to be used are of great importance. Depending on what dimension of parallelism is chosen the demands on the communication will be different. We review the major communication principles and comment on their use in ANN implementations.

6.1. Communication by Broadcast

In most parallel computers the most efficient way to communicate is to broadcast, since so many destinations receive information simultaneously. In fact, in SIMD computers broadcast is also used to distribute control information. As shown above, training example parallel-

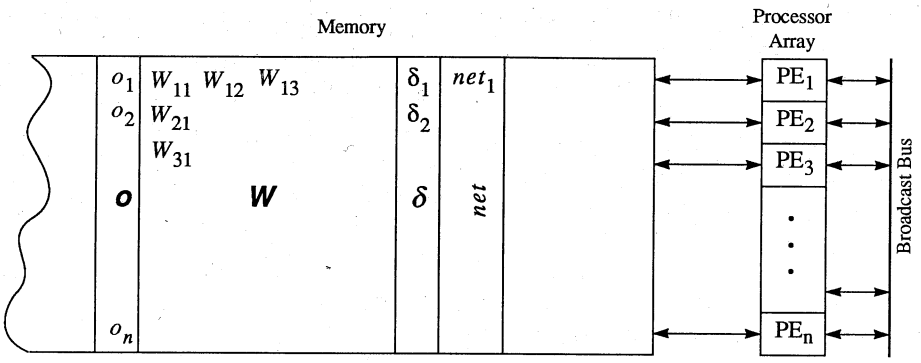

FIG. 8. Mapping of node parallelism into a processor array.

ism has a very large amount of possible parallelism. Since the basic communication needed with this form of parallelism is broadcasting it is a good choice if the maximum speed (CUPS) is to be obtained. If the batch or epoch type of weight updating cannot be used, node or weight parallelism must be used. With the required communication patterns in those forms of parallelism it is less obvious how broadcast can be used, and therefore it is harder to obtain maximum performance.

Using node parallelism, perhaps the most natural way to map the forward pass of a BP calculation on a processor array is to see it as a matrix–vector multiplication and map each row of W into one PE; see Fig. 8. In each step of the multiplication process, one PE broadcasts its node activation o_i to all other nodes. Each PE then multiplies the value with a weight value from the corresponding column of W and adds the result to a running sum net_i. After all activation values have been sent and multiplied, each PE holds one element of the resulting vector. In the backward phase, summation is required across the PEs instead of within each PE. This is slow unless another communication structure is added, e.g., an adder tree as proposed in [124]. Mapping in this way may also introduce inefficiency problems if the layers are of very different sizes. It is also difficult to utilize any sparsity in the connections with this method.

6.2. Grid-Based Communication

A natural way to arrange the communication required for weight parallelism is grid-based communication suitable for two-dimensional arrays of PEs. The PEs of the top edge, say, correspond to the source node layer, and the PEs of the right edge, say, correspond to the destination node layer. The weight matrix is distributed over the rest of the PEs. The input layer nodes at the top send their values down over the matrix by vertical broadcast. Then there is a horizontal summation phase. This is then repeated for the next layer, first horizontally and then vertically.

This scheme has been used or suggested by, e.g., Singer [120] and Fujimoto and Fukuda [33].

6.3. Communication by Circulation

One way to solve the communication when node parallelism is used is by a "systolic" ring structure [52, 70–72, 100]; see Fig. 9. In the forward phase (upper part of the figure) the activation values are shifted circularly along the row of PEs and multiplied with the corresponding weight values. Each PE accumulates the sum of the products it has produced. In the backward phase (lower part of the figure) the accumulated sum is shifted instead.

This scheme shares the possible inefficiency problems with the broadcast-based scheme (see end of Section 6.1).

6.4. Communication by General Routing

Some massively parallel computers, e.g., the Connection Machine, provide support for general routing by packet switching. Utilizing this facility is a straightforward method on these computers, but, since general routing normally is much slower than regular communication, the ANN computations will be communication bound, maybe with the exception of sparsely connected networks.

An implementation of directed graphs on computers lacking general routing capability has been suggested by Tomboulian [128]. It relies on SIMD computers with very modest communications facilities. Sending a value or a message between two nodes amounts to routing the message from PE to PE. A send is divided into time slots where each node knows if or where it should pass the current incoming message. There are means to extend the communication pattern dynamically which makes it very attractive for networks that use structural adaptation, like Fahlman and Lebiere's "Cascade Correlation" [24] or Kassebaum *et al.*'s [60] and Tenorio and Lee's self-organizing neural network (SONN) algorithm [125, 126].

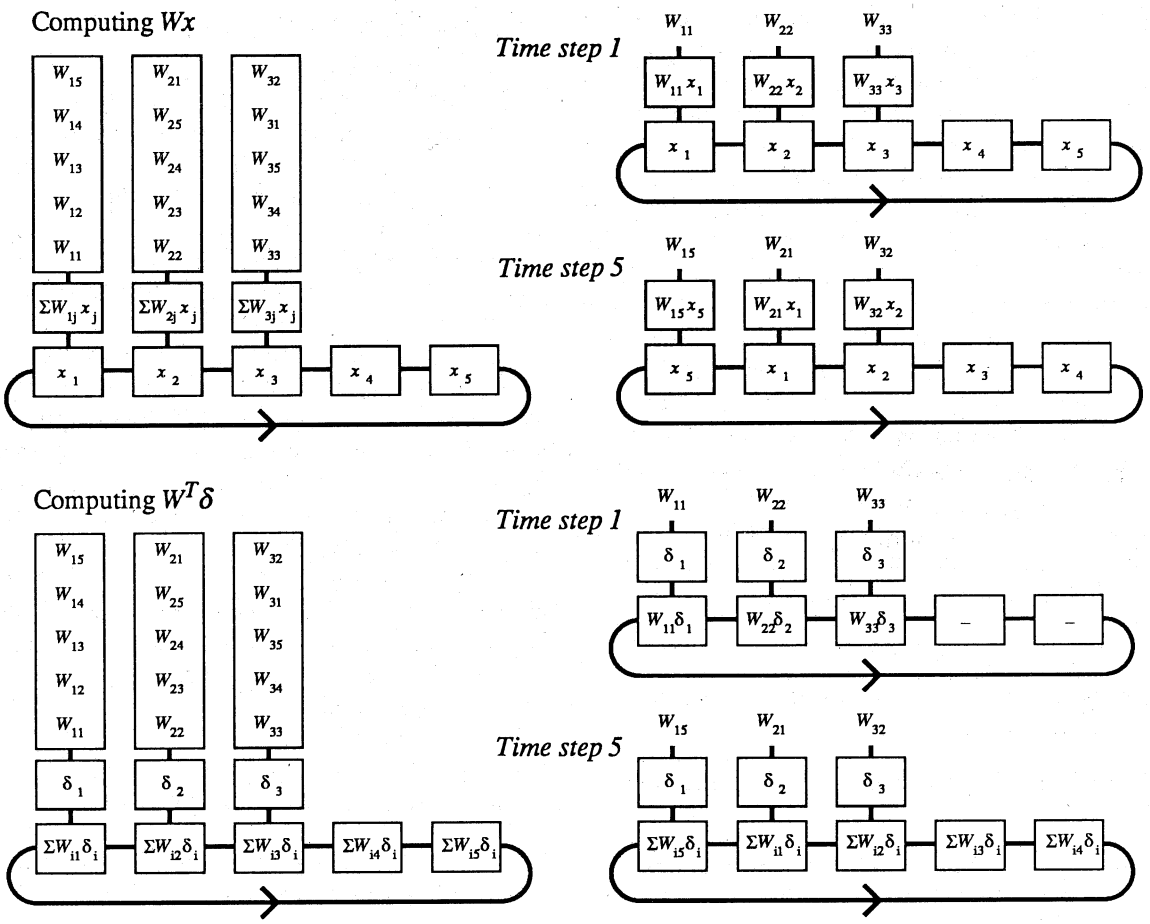

FIG. 9. Forward (top) and backward (bottom) phase of a BP calculation on a systolic ring processor.

Tomboulian's method has for dense networks been found to consume a great deal of PE memory for its tables [128]. For sparse networks the time and memory requirements are proportional to the product of the average number of connections per neuron and the diameter of the array.

6.5. Comments on Communication

For all communication strategies except the general routing methods, sparse connection matrices will lead to underutilization of the computer's PE resources. Lawson *et al.* have addressed this problem with their "SMART" (sparse matrix adaptive recursive transform) machine [73]. By using tables and special hardware they can make use of zero-valued connections to reduce simulation size and time. The communication is based on a ring structure together with a bus. However, Lawson's solution is not directly applicable on a massively parallel SIMD computer.

The massive flow of data into and out of the array, which will be the case in practical real-time applications and also in out of core calculation (very large simulations), places specific demands on the communication structure. In the design of a massively parallel computer for ANN, the PEs should be optimized for the typical operations, in order not to make the processing computation bound. Furthermore, the interconnection network should be optimized for the computational communication pattern, in order not to make the processing communication bound. Finally, the I/O system should be optimized to suit the needs of the application, in order not to make the system I/O bound. So far not very much attention has been paid to the latter problem complex, but it is certainly very much connected with the communication matters.

Training example parallelism is the form of parallelism that puts the weakest demands on communication. However, it is not interesting in real-time applications or in pure recall situations.

7.0. MEASURING THE SPEED OF ANN SIMULATIONS

In order to compare different implementations, some kind of standard measuring procedure is needed. The

number of multiply-and-add operations per second of which the computer is capable might be considered for such a measure, since this operation was identified as the most important one in most of the algorithms. However, it can serve only as an upper limit for performance; i.e., it marks the lowest speed that the computer manufacturer guarantees will never be exceeded. The utilization of this potential may be very different for the various ANN models and for different problem sizes.

Some of the commonly used indications of speed will be given below. They are of two kinds: first, there are general speed measurements, similar to the multiply-and-add performance measure. In order for these to be of any real value, the ANN model and problem size for which they were measured should always be given as additional information. Second, there are benchmarks, i.e., commonly implemented applications of specific size. The area is still too young (i.e., in the present "wave") for benchmarks to be very well developed.

7.1. Measurements

Some measurements commonly used in the ANN community are the following.

7.1.1. CPS or IPS—Connections (or Interconnections) per Second

Each time a new input is provided to an ANN, all connections in the network must be computed. The number of connections per second that the network can perform is often used as a measure of performance. When measuring CPS, computing the nonlinear thresholding function should be included. Comparing the CPS measure to the multiply-and-add measure roughly indicates how well the specific algorithm suits the architecture. When comparing CPS values for different implementations, the precision of calculation is important to consider. Other things which may influence the CPS measure are the problem size and the choice of nonlinear thresholding function.

7.1.2. CUPS or WUPS—Connection (or Weight) Updates per Second

The CPS number only measures how fast a system is able to perform mappings from input to output. To indicate the performance during training a measurement of how fast the connections can be updated is given as a CUPS figure. In a back-propagation network both the forward and the backward passes must be computed. Typically the CUPS number is 20–50% of the CPS number.

For self-organizing maps CUPS have been used as the number of connections in the network multiplied by the number of network updates per second, despite the fact that very few connections are actually updated when using small neighborhoods at the end of a training session.

7.1.3. Epochs

An epoch is defined as a single presentation of each of the examples in the training set, in either fixed or random order. The concept is well-defined only for problems with a fixed, finite set of training examples. Modification of weights may occur after every epoch, after every pattern presentation, or on some other schedule.

The epochs concept is sometimes used when measuring the number of times one must run through the training set (one run is one epoch) before some predefined condition is met. This is used to compare different versions of an algorithm or different algorithms. When doing this one should be aware of the fact that the computational effort to go through an epoch may vary considerably from one algorithm to another. Even if an algorithm learns in half the number of epochs it can still take longer time because the calculations are more complicated.

An epochs per second measure may also be used as an alternative to CUPS to indicate the speed of learning. It gives a number that is easier to grasp than the CUPS number. Of course the measure is strongly related to the problem and training set used.

For problems where an epoch is not well defined, learning time may instead be measured in terms of the number of individual pattern presentations.

7.1.4. CPSPW—Connections per Second per Weight (or SPR—Synaptic Processing Rate)

A measure that indicates the balance between processing power and network size (i.e., number of weights) has been introduced by Holler [46]. His argument for the importance of this measure is the following: Biological neurons fire approximately 100 times per second. This implies that each of the synapses processes signals at a rate of about 100 per second; hence the SPR (or, to use Holler's terminology, the CPSPW) is approximately 100. If we are satisfied with the performance of biological systems (in fact, we are even impressed by them) this number could be taken as a guide for ANN implementations. Many parallel implementations have SPR numbers which are orders of magnitude greater than 100, and hence have too much processing power per weight. A conventional sequential computer, on the other hand, has an SPR number of approximately 1 (if the network has about a million synapses); i.e. it is computationally underbalanced. It should be noted that Holler's argument is of course not applicable to batch processing training situations.

7.2. Benchmarks

There are some commonly used problems that may be considered benchmark problems. They are used in a number of different ways, e.g., to measure learning speed, quality of ultimate learning, ability to generalize, or combinations of these factors. Thus their use is not

restricted to speed comparisons. On the contrary, most of the benchmarks have been introduced to compare algorithms and not architectures. This means that most of the benchmarks should not be used to compare suitability of architectures for the simulation of ANNs. Among such benchmark problems are the XOR problem, the Parity problem, and the Two Spirals problem [22]. They are all small problems intended for qualitative comparison of algorithms.

7.2.1. NETtalk

A larger and more realistic application is known as NETtalk, a text to phoneme translation solved by a back-propagation network, described by Sejnowski and Rosenberg [114]. The task is to train a network to produce the proper phonemes, given a string of letters as input. This is an example of an input/output mapping task that exhibits strong global regularities, but also a large number of more specialized rules and exceptional cases. It is often used as a benchmark.

The experimental setup used by Sejnowski and Rosenberg [114] described by Fahlman in [22] was the following: The input to the network is a series of seven consecutive letters from the training text. The central letters in this sequence is the "current" one for which the phonemic output is to be produced. Three letters on either side of this central letter provide a context that helps to determine the pronunciation. Of course, there are a few words in English for which this local seven-letter window is not sufficient to determine the proper output. For the study using this "dictionary" corpus, individual words are moved through the window so that each letter in the word is seen in the central position. Blanks are added before and after the word as needed. Some words appear more than once in the dictionary, with different pronunciations in each case; only the first pronunciation given for each word was used in this experiment.

A unary encoding is used. For each of the seven letter positions of the input, the network has a set of 29 input units: one for each of the 26 letters in English, and three for punctuation characters. Thus, there are $29 \times 7 = 203$ input units in all. The output side of the network uses a distributed representation for the phonemes. There are 21 output units representing various articulatory features such as voicing and vowel height. Each phoneme is represented by a distinct binary vector over this set of 21 units. In addition, there are 5 output units that encode stress and syllable boundaries. Typically 60 to 120 hidden units have been used.

In the absence of very large benchmarks, we will sometimes use figures on NETtalk to compare architectures (if they have been reported). Otherwise we will report the implemented network structure and training method together with the performance measure given.

8.0. CHARACTERIZATION OF COMPUTER ARCHITECTURES

As the structure of ANN algorithms is naturally parallel it is relatively easy to make use of a very large number of PEs. Computers with a very large number of PEs are often called massively parallel. Being massive should mean that the structure gives an impression of being solid. That is, the number of units should be so high that is impossible to treat them individually; they must be treated *en masse*. By definition then, each PE must be told what to do without specifying it individually. This is actually the concept of the SIMD or SCMD computer (defined later).

The lower bound for massive parallelism will then be set by the largest computer in which each PE is treated as an individual with its own instruction flow (MIMD). We think that for the moment $2^{12} = 4096$ is a suitable limit.

An interesting extension to massively parallel is the concept of *continuously* parallel. This should mean the limit of massively parallel as the number of processing elements becomes infinite [77].

It is useful to have characterizations also of the lower degrees of parallelism. To get a reasonable "definition" with a nice symmetry we suggest a rough division between *highly* parallel, *moderately* parallel, and *barely* parallel. By defining the limits to 2^{12}, 2^8, 2^4, 2^0 we have an easy-to-remember scheme for the characterization. When appropriate, in the future the limits may be moved upward. Summarizing this we get the following "definitions" which will be used in this paper (N stands for the number of PEs):

Continuously parallel	$N \to \infty$
Massively parallel	$N \geq 2^{12}$
Highly parallel	$2^8 \leq N < 2^{12}$
Moderately parallel	$2^4 \leq N < 2^8$
Barely parallel	$2^0 < N < 2^4$

This characterization is completed with an "orthogonal" one describing the computational power of the PEs. The power or complexity can of course be measured in many ways but as a coarse measure we use the bit-length of the natural data type for the processing elements. These two characterizations result in the diagram shown in Fig. 10.

We will concentrate on the massively and highly parallel architectures in the next section. But we will also, for comparison, include some moderately and even barely parallel computers like Warp systems with more complex control. This is because the use of these computers has given interesting results with respect to algorithms and ways of mapping ANNs to a parallel computer. It should be noted that many of those algorithms do not use the powerful control, but instead use a SIMD or SCMD structure.

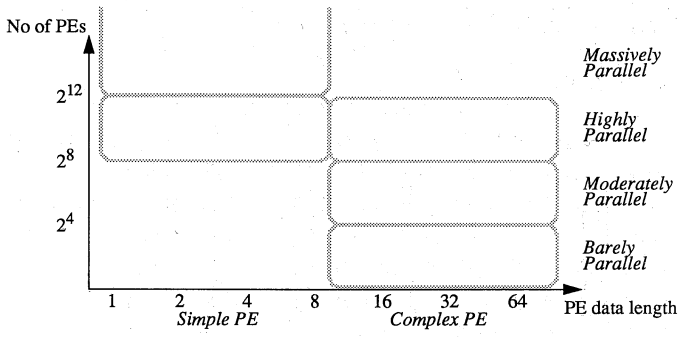

FIG. 10. Classifying architectures for ANN by dividing according to the number of and complexity of the processing elements that are used.

Equally important as the degree of parallelism is the organization of the computer. The much used characterization due to Flynn [28] divides the computers into groups according to the number of instruction streams and the number of data streams. Of interest for ANN computations are the groups with multiple data streams: SIMD (Single Instruction stream, Multiple Data streams) and MIMD (Multiple Instruction streams, Multiple Data streams).

To characterize a MIMD computer used as a SIMD architecture, SCMD (Same Code for Multiple Data streams) is suggested and used in this paper.

8.1. Division of SIMD

The SIMD category in itself shows great architectural variations. We briefly review some of the major groups.

8.1.1. Systolic Arrays

Systolic arrays represent a general methodology for mapping high-level computations into hardware structures. Developed at Carnegie Mellon by Kung and others [68], the concept relies on data from different directions arriving at cells/PEs at regular intervals and being combined. The number of cells in the array and the organization of the array are chosen to balance the I/O requirements.

Systolic architectures for ANN have been proposed by, among others, Hwang *et al.* [52] and Kung and Hwang [70, 71]. They have found that a ring structure or cascaded rings are sufficient (cf. Section 6.3 on communication by circulation).

8.1.2. Linear Processor Arrays

In this group are the processor arrays with the simplest structure, the linear (one-dimensional) arrays. The linear structure is often combined with the ring structure and the bus structure (i.e., broadcast). Actually, the arrays in this group are typically not massively parallel due to the limitations of the communication structure. Some of the arrays can be scaled to at least a few thousand processors. They can however be strung together with similar arrays and form a multiple SIMD array which can be much larger than any single SIMD module.

8.1.3. Mesh-Connected Processor Arrays

In silicon the planar structure of the connecting medium will tend to favor planar communication networks. Therefore, it is natural to build the communication on a mesh or a grid; i.e., each PE has four (or eight) neighbors. Even the computers with multidimensional communication have included mesh connections for faster local communication. Examples of mesh-connected arrays are DAP (Distributed Array Processor) [51], MPP [101], and BLITZEN [9].

8.1.4. Multidimensional Processor Arrays

Multidimensional architectures like hypercubes allow a more general communication pattern than any of the previous arrays. That is, no PE is further away than $\log_2 N$ steps, where N is the number of PEs.

It has been found that general communication is used mainly for transfers between "parameter spaces" (e.g., image → edges → corners). None of the efficient implementations of ANN algorithms on any of the studied architectures used/needed multidimensional communication.

9.0. PARALLEL COMPUTERS DESIGNED OR USED FOR ARTIFICIAL NEURAL NETWORKS

Placing some of the parallel computers used for ANN into the diagram of Fig. 10 results in the map shown in Fig. 11.

We will now give brief descriptions of these machines and review the reported ANN implementations. We certainly do not cover all the massively and highly parallel machines but our selection should be representative. The order of presentation is alphabetical within each group.

9.1. Massively Parallel Machines with Simple PEs

9.1.1. AAP-2

AAP-2 [132] is a two-dimensional (2D) mesh-connected computer enhanced with bypasses and ripple through operations. It contains 64K (256 by 256) bit-serial PEs featuring a simple 1-bit ALU and a 144-bit register file. There is also a 15-bit control register which can control each PE individually in a primitive way.

BP has been implemented in a node and weight paral-

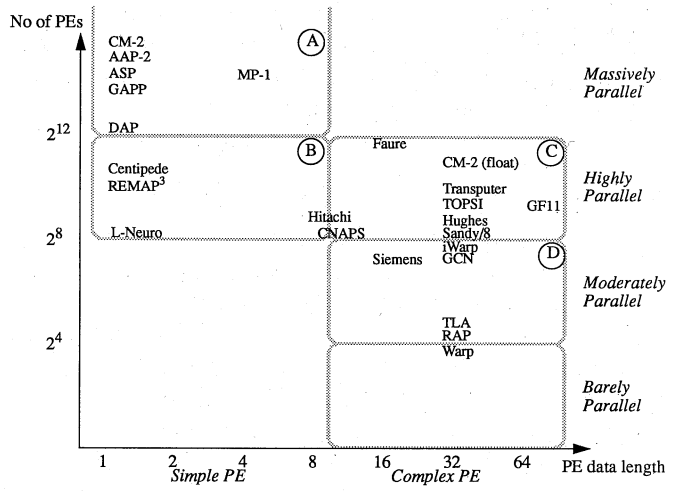

FIG. 11. A way of classifying architectures for ANN. A corresponds to massively parallel machines with simple PEs in Section 9.1, B to highly parallel machines with simple PEs in Section 9.2, C to highly parallel machines with complex PEs in Section 9.3, and D to moderately parallel machines with complex PEs in Section 9.4.

lel fashion using an interesting circular way to perform table lookup of the sigmoid function. Using 26 bits for weights and activations and a 256 × 256 × 256 network 18 MCUPS was achieved. When different sizes of the layers were used the efficiency decreased.

9.1.2. *Associative String Processor (ASP)*

The ASP is a computer designed and built by University of Brunel and Aspex Ltd. in England, with Lea as the principal investigator [74]. The computer consists of simple PEs (1-bit full adder, $n + a$-bit comparator) which are strung together into "strings" of more "complex" types such as 32-bit integers. Each processor has a 96-bit data register and a 5-bit activity register. Each string is linked by a communication network and there are data exchanges and buffers to support high-speed data I/O. The architecture is expandable up to 262,144 processors arranged as 512 strings of 512 processors each. A 16K machine has been built and a 64K machine is to be completed by the summer of 1992.

Krikelis and Grözinger [67] have implemented Hopfield net and BP on a simulator of the architecture. In the Hopfield net one neuron is mapped onto each PE as long as the internal memory is sufficient (1800 nodes). At this maximum, the real machine should be able to run at 1600 MCPS.

The BP network implementation also uses node and weight parallelism simultaneously. On a 63 × 45 × 27 network 4323 PEs are utilized, but in some time instances there are only 46 PEs actively taking part in the calculation. The weights are represented by 16-bit fixed-point numbers. A thresholded picture (1 bit deep) is used as input and the activation in the other nodes is represented by 12 bits. The sigmoid is approximated with a group of conditionals. The simulation indicates a performance of 12 MCUPS on the real machine.

9.1.3. *Connection Machine (CM, CM-2)*

The Connection Machine [40, 127] manufactured by Thinking Machines Corporation (TMC) is for the moment the most massively parallel machine built (from 8K up to 64K processing elements). In addition to its large number of processors, two of its strong points are its powerful hypercube connection for general communication and the multidimensional mesh connection for problems with regular array communication demands. In the CM-2 model TMC also added floating-point support, implemented as one floating-point unit per 32 PEs. This means 2048 floating-points units on a 64K machine, giving a peak performance of 10 GFlops.

CM-2 is one of the most popular parallel computers for implementing ANN algorithms. Most of the implementations so far concern BP, but within this model different forms of parallelism have been utilized in different applications.

Rosenberg and Blelloch [108] constructed an algorithm in which each neuron is mapped onto one PE and each weight (synapse) is mapped onto two PEs. This unusual mapping is chosen in order to use some special communication features of the Connection Machine. Their mapping could be seen as a general method for mapping directed graphs onto a computer with "copy scan," "send," and "plus-scan" operations. The resulting performance, limited solely by communication, was 2.8 MCUPS for NETtalk and around 13 MCUPS maximally. Forward–backward parallelism was mentioned but not implemented.

Brown [12] compared two different ways of parallelizing BP. One method used node parallelism with one PE per neuron and the other was the method (node plus weight parallelism) suggested by Rosenberg and Blelloch. Brown found that the latter had better performance.

Zhang *et al.* [139] combined training example and node parallelism plus good knowledge of the CM-2 communication and computational structure to achieve high performance on quite different sizes of ANNs. On NETtalk using 64K PEs they could get around 40 MCUPS (and 175 MCPS).

Using Zhang *et al.*'s approach on a much larger problem (280.5 megabyte of training examples) Diegert [19] reached 9.3 MCUPS on a 16K PE machine. With 64K PEs the estimated performance is 31 MCUPS. This is good in comparison with the NETtalk performance above, when considering that the training data are moved in and out of the secondary storage all the time.

Singer's implementation of BP [118, 119], which is the fastest implementation on the CM-2, uses training example parallelism. He reports a maximum of 1300 MCPS and 325 MCUPS on a 64K PE machine when using 64K training vectors.

Deprit [18] has compared two implementations of recurrent back-propagation (RBP) on the CM-2. First he used the same mapping as Rosenberg and Blelloch (R&B) with a minor modification to include the feedback connections. The second mapping was node parallelism with the communication method suggested by Tomboulian [128]; see Section 6.4. The basic finding was that the R&B method was clearly superior for densely connected networks, such as that used in NETtalk. Note that the R&B implementation is still communication bound, indicated by the fact that the simulation time changed almost imperceptibly when software floating point was used instead of the hardware units.

Obermayer *et al.* have implemented large SOM models on the CM-2 [91]. Node parallelism with up to 16K PEs (neurons) was used. The input vector length (input space dimension) was varied and lengths of up to 900 were tested. The same algorithm was also implemented on a self-built computer with 60 T800 (Transputer) nodes connected in a systolic ring. In addition to algorithmic analysis the two architectures were benchmarked. The conclusion was that the CM-2 (16K PEs) with floating-point support is equal to 510 Transputer nodes for the shortcut version of SOM. As a 16K CM-2 has 512 Weitek FPUs, each with approximately the same floating-point performance as one T800, it can be concluded that the shortcut method is basically computation bound. In a "high-communication" variant of SOM, a 30-node Transputer machine would run at one-third of the CM-2 speed.

Rogers [105] has used CM-2 as a workbench for exploring Kanerva's SDM model. Rowwise mapping (node parallelism) is used for the selection phase (steps 1 and 2 in Algorithm 3), but for the store/retrieve phase weight parallelism is used (as many PEs as there are counters). As the number of physical PEs in Rogers' implementation is of the same order as the number of counters in one column he actually uses node parallelism and rowwise mapping, letting the CM-2 sequencer take care of the looping over each column. Implementing this in *Lisp on an 8K CM-2 results in a performance of only approximately 3 iterations per second (256-bit address, 256-bit data, 8192 locations). Using a pure rowwise mapping in C* one of the present authors has been able to achieve between 30 and 70 iterations per second on a CM-2 of this size. The difference is probably due to some unnecessary but expensive communication needed in going from 1D to 2D representation.

The still relatively poor performance of SDM on CM-2 is found to depend on at least three factors: PEs are underutilized during the select/retrieve phase using node parallelism where as few as 0.1–1% of the total number of PEs are active; the natural rowwise mapping demands time-consuming sum-reduction across PEs; the "optimal" mixed mapping [89] (see Section 9.2.5) is hard to implement efficiently with the current sequencer.

9.1.4. DAP

The Distributed Array Processor—produced by ICL (International Computers Limited) and AMT (Active Memory Technology Ltd.) [51]—is a series of 2D SIMD computers with different sizes and different hosts. The sizes range from 1024 to 4096 PEs. The processing elements are simple and bit-serial. To overcome the shortcomings of the ordinary 2D array (its long distance communication performance) row and column "highways" have been included. A 4K DAP gives 32-48 Mflops and computes 8-bit multiplications at 250 Mops and 8-bit multiplications by scalar constant at 600 to 1200 Mops. In a forthcoming machine there will be some support for multiplication in each PE. It will then give 560 MFlops maximally for a 4K PE machine.

Forrest *et al.* [29, 30] report the use of the ICL DAP to implement Hopfield net, BP, Elastic net (applied to traveling salesman problems), etc. They also use Transputers for similar tasks. However, no performance figures are given; it is more of a feasibility study. The authors describe and use four of the various types of parallelism: node, weight, training example, and training session.

Núñez and Fortes [90] have used the AMT DAP to implement BP, recurrent BP, and mean field theory. The calculations are treated as matrix operations, which with our terminology results in a combination of weight and node parallelism. For activation distribution the DAP broadcast highways are utilized in a way similar to the grid-based communication method. On a 4K machine with 8 bits used for weight and activation the performance is 100–160 MCUPS for BP. With 16 bits used instead, the figures are 25–40 MCUPS. A NETtalk (203 $\times$ 64 $\times$ 32) implementation on the 1K DAP using 8 bits resulted in 50 MCUPS.

9.1.5. MasPar (MP-1)

MasPar MP-1 [8, 15, 87] is a SIMD machine with both mesh and global interconnection style of communication. It has floating-point support, both VAX and IEEE standards. The number of processing elements can vary between 1024 and 16,384. Each PE has forty 32-bit registers, a 4-bit integer ALU, floating-point "units" to hold Mantissa and Exponent, an addressing unit for local address modifications, and a 4-bit broadcast bus.

MP-1 has a peak performance, for a 16K PE machine, of 1500 MFlops single-precision [87]. It is programmed in parallel versions of Fortran (MasPar Fortran) or C (MasPar C) or a C-derived language (MasPar Application Language) for more direct contact with the hardware [15].

Chinn *et al.* [14] and Grajski *et al.* [35, 36] have implemented BP and SOM using floating-point arithmetic. Estimating the performance of a 16K PE machine from the figures of a 2K, 4K, and 8K machine gives approximately 10 MCUPS with BP on a 256 × 128 × 256 network, utilizing node and weight parallelism. The mapping of SOM into MP-1 is one PE to each SOM node. It was measured to give 18 MCUPS on a 4K machine when 32-dimensional input vectors (or larger) were used.

9.2. Highly Parallel Machines with Simple PEs

9.2.1. AIS-5000, Centipede

AIS-5000 [113] manufactured by Applied Intelligent Systems Corp. has up to 1024 bit-serial PEs arranged as a linear array. The PEs are similar to those of CM, DAP, and BLITZEN. The basic problem when using AIS-5000 for ANN is the lack of support for multiplication and the difficulties in performing the backward phase of back-propagation (BP). Both of these difficulties have been addressed in [124] for other linear processor arrays. In a new generation using the new Centipede chip [135] better support for multiply-and-add is incorporated and table lookup is also possible.

AIS-5000 is intended mainly for image processing applications like inspection, but Wilson [135] has shown a neural network implementation of feedback type (Hopfield). Despite the difficulties mentioned, between 19 and 131 MCPS (depending on the precision used) is achieved using node parallelism.

9.2.2. Connected Network of Adaptive ProcessorS (CNAPS)

CNAPS, manufactured by Adaptive Solutions, Inc., is one of the first architectures developed especially for ANN. Called X1 in the first description by Hammerstrom [37], it is a 256-PE SIMD machine with a broadcast interconnection scheme. Each PE has a multiply (9 × 16 bit)-and-add arithmetic unit and a logic-shifter unit. It has 32 general (16-bit) registers and a 4-kbyte weight memory. There are 64 PEs in one chip. The user may choose 1, 8, or 16 bits for weight values and 8 or 16 bits for activation values.

With 16 bits for weights and 8 bits for activation the peak performance is 5 GCPS for a feedforward execution, and 1 GCUPS using BP learning. For NETtalk 180 MCUPS is achieved using only one chip (64 PEs).

The performance of CNAPS on SOM was reported by Hammerstrom and Nguyen [38]. The figures are based on a 20-MHz version of CNAPS. With 512 nodes/neurons and a 256-dimensional input vector, best match using a Euclidean distance measure can be carried out in 215 μs, using 16-bit weights. Making their CUPS figure comparable to others, the performance is about 183 MCUPS.

9.2.3. Geometric Arithmetic Parallel Processor (GAPP)

GAPP is a mesh-connected SIMD systolic array. On one chip there are 72 PEs (6 × 12), each with 128 bits of RAM. Designed for image processing and working bit-serially, it runs at a clock speed of 10 MHz. The processing element is very simple, basically a full adder. The chip was developed by Holsztynski of Martin Marietta Aerospace Corp. and manufactured by NCR Corp [16]. It was the first commercially available systolic chip.

Brown *et al.* have used GAPP to implement BP [11]. As there is no floating-point support on GAPP they use fixed-point numbers. Ten bits precision is used for the activation values and 15 bits (4 + 11) for the weights. The sigmoid function is approximated by a stepwise linear function. Both weight and node parallelism are used. No performance figures are reported.

Barash and Eshera [5] have also used GAPP to implement feedforward networks with BP learning. Using a variation of communication by circulation described in Section 6.3 they combine weight and node parallelism. With a 40K GAPP and 8 bits for weight and activation values they estimate a performance of 300 MCUPS. The major bottleneck is reported to be the calculation of the sigmoid function.

9.2.4. L-Neuro

The Laboratoires d'Electronique Philips (LEP), Paris, have designed a VLSI chip called L-Neuro. It contains 16 processors working in SIMD fashion. In association with these chips Transputers are imagined as control and communication processors. Weights are represented by two's-complement numbers over 8 or 16 bits, and the states of the neurons are coded over 1 to 8 bits. The multiplication is done in parallel over 16 or 32 weights but bit-serially over the weight values, and as only one output node at the time is calculated (in one chip) it can be considered a weight parallel implementation. The node activation value must go outside the chip for distribution to other nodes, and no support for the calculation of the sigmoid function is needed/implemented inside the chip. Duranton and Sirat [20, 21] have described implementations of SOM, Hopfield, and BP networks using this chip as a building block.

9.2.5. REMAP³

REMAP³ is a cooperative research project between Luleå University of Technology and Halmstad University, both in Sweden. The project is aimed at obtaining a massively parallel computer architecture put together by modules in a way that allows the architecture to be adjusted to a specific application. This suggests that a certain architecture may be "compiled"; thus a modification of each module and adjustments of the connections be-

tween the modules are enforced. The intended application area in a broad sense is embedded industrial systems. A multimodule system is well suited for implementing a multiple-network artificial neural system.

A small prototype of a software configurable processor array module with bit-serial processors has been implemented [75] and a larger system consisting of several modules is in the process of being designed. Different variations can be realized by reprogramming. An architecture tuned for neural network computations, including a fast bit-serial multiplier, has been designed. Åhlander and Svensson [140] describe how support for floating-point calculations may be embedded in the bit-serial working mode if needed, resulting in impressive floating-point performance when implemented with VLSI.

The mapping and performance of some common ANN models (BP, Hopfield, SOM, SDM) have been reported [124, 88, 89]. As an example, a 4K PE machine reaches 953 MCPS or 413 MCUPS on BP when running at 10 MHz and using 8-bit data. A node parallel version of BP is used. For 16-bit data, 546 MCPS or 219 MCUPS is achieved. An SDM model running 30 iterations per second on a 8K CM-2 can run at speeds above 400 iterations per second on a 256-PE REMAP^3 (10 MHz) with counters. The implementation uses a "mixed mapping" of the SDM model: rowwise mapping for the selection phase and columnwise for the store/retrieve phase.

9.3. Highly Parallel Machines with Complex PEs

9.3.1. GF11

GF11 is an experimental SIMD computer built at IBM Research Laboratory [7]. It has 566 PEs running at 20 MHz and a peak performance of 11 GFlops (as the name implies). It is intended primarily for quantum chromodynamics (QCD) predictions. Each PE has two floating-point adders, two floating-point multipliers, and one fixed-point unit. Table lookup and selection are the only data-dependent operations available.

The memory is organized as a three-staged hierarchy of progressively larger and slower memories. The communication is done via a three-staged Benes network. The machine is programmed in conventional C with calls to special-purpose procedures. This generates, after an optimization step, large blocks of microcode. Using a very simple controller the code is sent to all processors (there is also an address generator/relocator).

Witbrock and Zagha have been able to use this computer, before it was completed, to run ANN algorithms [136]. They implemented BP and benchmarked it with the NETtalk text-to-speech benchmark, achieving a speed of 900 MCUPS on a 356-processor computer. Because of the memory hierarchy they needed a few tricks to obtain this high speed.

Witbrock and Zagha discuss various ways to parallelize BP and finally choose training example parallelism. When all the weight changes are added, $\log_2 N$ steps are required. They also discuss in detail how to compute the sigmoid function and how to implement Recurrent BP because this model is their main interest. They conclude that a sophisticated communication network is not necessary if processor-dependent addressing is available.

9.3.2. Hughes Systolic/Cellular Architecture

The Hughes machine is a mesh-connected SIMD architecture made for image formation of synthetic aperture radar (SAR) pictures. The prototype used has 256 PEs, each with seven function units working on 32-bit fixed-point data (two multipliers, two adders, a divider, a normalizer, and a sorter), 24 memory registers, and a small local memory.

On this computer Shams and Przytula have implemented BP [115]. Training example parallelism was used in one "dimension" of the mesh and node parallelism with communication by circulation in the other dimension. Benchmarking with NETtalk resulted in a performance of 100 MCUPS, including loading and unloading operations. For recall only the result was 240 MCPS.

9.3.3. UCL Neurocomputer

Treleaven *et al.* have set out to construct a general-purpose "neurocomputer" [129]. Each PE is a 16-bit RISC (16 instructions) containing an on-chip communication unit and on-chip local memory. For communication a ring structure and a broadcast bus are used. Each PE has a unique address to support a more general type of logical communication. The ALU can add and subtract in one clock cycle but it only supports multiplication with a multiply step. This means that the performance of each PE will be low on ANN problems that have many multiplications.

The 5-MHz, 1.5-μm CMOS chip which was ready in 1990 could contain only two PEs and have a maximum CPS rate of 156 kCPS/PE or 312 kCPS/chip. It looks like the complex control part of the chip made it difficult to include a more powerful ALU with a one-cycle multiply-and-add. The UCL approach should be compared with the CNAPS approach in which emphasis is placed on the basic computation of ANNs and the controller is shared between all the PEs (SIMD).

9.3.4. Transputers

The Transputer [54, 133] is a single-chip 32-bit microprocessor. It has support for concurrent processing in hardware which closely corresponds to the Occam [10, 53, 55] programming model. It contains on-chip RAM and four bidirectional 20 Mbits/s communication links.

By wiring these links together a number of topologies can be realized. Each Transputer of the T800 type is capable of 1.5 MFlops (20 MHz) and architectures with up to 4000 Transputers are being built [133].

The next generation of Transputers, called T9000, will provide around 10 sustained MFlops and have 16 kbyte of cache memory on chip. The communication has also been improved to support through-routing without processor involvement. This will be an even better building block for highly parallel computers for the nineties.

Back-propagation has been implemented by Hwang *et al.* [52, 71] and Petkov [98] using node parallelism and communication by circulation. Forrest *et al.* have implemented the Hopfield, BP, and Elastic net models on both DAP and Transputers [29, 30]. No figures of performance have been given, but Transputers with their relatively low communication bandwidth (relative to their computational capabilities) are more efficiently used if small-grain partitioning can be avoided. That is, node and weight parallelism should be avoided, especially if they are described as general graphs.

The SOM of Kohonen has been implemented by Hodges *et al.* [45] and Siemon and Ultsch [117]. Both implementations just distribute the nodes over the PEs and use ring communication to distribute the input vector and to find the global minimum. As long as the neighborhood is larger than the number of PEs this mapping is quite efficient. Good performance will be achieved also for high input dimension. Siemon and Ultsch state a performance of 2.7 MCUPS on a 16 Transputer machine applied to a network sized 128 × 128 using 17 input dimensions. Hodges *et al.* presented an equation for the performance but no concrete implementation.

A more general implementation framework, called CARELIA, has been developed by Koikkalainen and Oja [66]. The neural network models are specified in a CSP-like formalism [64–66]. The simulator is currently running on a network of Transputers and some of the models implemented are SOM, BP, and perceptrons. The performance of the simulator has not been reported.

The European PYGMALION project [130] is, like CARELIA, a general ANN programming environment which has Transputers as one of its target architectures. Its ANN programming languages are based on C++ and C; this together with a graphical software environment and algorithm libraries makes it a complete ANN workbench. Performance figures or an indication of how to use massively parallel computers as targets are unfortunately not given.

9.4. Moderately Parallel Machines with Complex PEs

9.4.1. DSP (Digital Signal Processor) Based

Because the current generation of DSPs and some of the RISC chips have outstanding multiply-and-add performance it is easy to conceive of them as building blocks for an ANN computer. There are at least five suggested architectures using i860, TMS320C40, or TMS320C30.

9.4.1.1. Sandy/8. Building Sandy/8, Kato *et al.* at Fujitsu [61] intend to use conventional processors or signal processors for the calculations and simple ring structures for communication. The system is projected to utilize 256 TMS320C30. The expected maximal performance using BP is above 500 MCUPS. Only 118 MCUPS is expected on NETtalk (203 × 60 × 26) as the mapping can utilize only 60 PEs.

9.4.1.2. Ring Array Processor (RAP). The RAP is a multi-DSP system for layered network calculations developed at the International Computer Science Institute, Berkeley, California [6, 83]. Each PE consists of a TMS320C30 connected to other PEs via a bus interface into a ring network. The 4-PE computer (one board) which has been implemented runs at maximally 13.2 MCUPS [82, 83]. A 16-PE system is estimated to run at 46 MCUPS.

9.4.1.3. GigaCoNnection (GCN). Hiraiwa *et al.* [43] at Sony Corp. are building an ANN computer called GCN in which each PE consists of a processor similar to the core of i860, 2 FIFOs, and 4 Mbyte RAM. Each PE will fit into a single chip (by 1992). The PEs are connected into a 2D mesh with wraparound. The ANN algorithm is mapped on the architecture using node parallelism in one direction (systolic ring) and training example parallelism in the other. The expected BP performance when running 128 PEs at 50 MHz will be above 1 GCUPS for a 256 × 80 × 32 network. The training data are then distributed into 32 groups.

9.4.1.4. TOPSI. TOPSI is a computer architecture built at Texas Instruments [31]. Each PE or module can be said to consist of a TMS320C40 and a 32-bit general processor like MC68040 together with support for communication. There are two complementary communication structures, one general reconfigurable inter-PE network and one hierarchical bus for broadcasting information. Only the latter is needed for the implementation of Hopfield and BP networks. A 100-module computer will run BP at a maximum speed of approximately 150 MCUPS, and a 1000 module computer at approximately 1.4 GCUPS.

9.4.1.5. PLANNS (Planar Lattice Architecture for Neural Network Simulations) and TLA (Toroidal Lattice Architecture. PLANNS is an improved version of the TLA, both suggested by Fujimoto and Fukuda [32–34]. They use node and weight parallelism with grid-based communication. Load balancing is achieved by first mapping the computations onto a virtual TLA and then splitting the work to suit a physical TLA. The physical

processor array must be able to support mesh communications.

A 16-PE Transputer array has been used as a prototype TLA resulting in 2 MCUPS on a feedforward network using BP. The authors claim an almost linear speedup with the number of PEs when their load balancing scheme is used. By using a more powerful PE like i860 and a larger number of nodes (some 30,000) they are planning to reach 10 GCUPS in a future implementation.

9.4.2. Warp

Warp is a one-dimensional array of 10 or more powerful processing elements developed at Carnegie Mellon in 1984–1987 [69]. Each cell/PE has a microprogrammable controller, a 5-MFlops floating-point multiplier, a 5-MFlops floating-point adder, and a local memory. Communication between adjacent cells can be conducted in parallel over two independent channels: a left-to-right X channel and a bidirectional Y channel. In 1991 Intel released a single-chip version of the Warp concept called iWarp [97]. Systems with up to 1024 iWarp chips can be built and can give a theoretical performance of 20 GFlops. Implementing a computer with these chips will at least double the performance figures given for Warp below.

Back-propagation was implemented by Pomerleau *et al.* [100] trying both node and training example parallelism. They found that with training example parallelism they could simulate much larger networks and/or at higher speeds. On NETtalk the 10-PE Warp reached 17 MCUPS.

An implementation of Kohonen SOM has been described by Mann and Haykin [79]. Using training example parallelism between 6 and 12 MCUPS was achieved. Some minor problems with the topology ordering process when using training example parallelism were reported. The authors suggest that either the network start at some order instead of at random state or the network be trained sequentially for the first 100–1000 steps, after which the training example parallelism is "turned on."

9.5. Other High-Performance Architectures

9.5.1. Vector (Super) Computers

For the sake of comparison with well-known powerful computers of a more conventional kind, some figures from implementations on a couple of, so-called, supercomputers are given.

9.5.1.1. CRAY X-MP. The performance on a Cray X-MP was given in the DARPA neural networks study [17] to be 50 MCPS. It can be compared to the theoretical maximal performance of 210 MFlops [44]. Even though the 50 MCPS performance is often cited, it is difficult to draw any conclusions from this, as the network size, the training algorithm, and even whether or not training is included are unknown.

9.5.1.2. NEC SX-2 and SX-X. NEC's SX-2 is a conventional supercomputer with four general vector pipelines giving a peak performance of 1.3 GFlops [44]. On ANN its performance is 72 MCUPS on NETtalk and its maximal performance on BP is 180 MCUPS [4] via [61].

9.5.2. VLSI Implementations

Even though the intention of this paper primarily is to review more complete computers, there are a few borderline cases like L-Neuro, UCL neurocomputer, and CNAPS. There are many other interesting suggestions and realizations of chips for ANN. More material and surveys can, for instance, be found in [17, 46, 84, 103, 112]. To review only the digital ones would lead to another paper of this length. Here we mention a few of the digital realizations:

9.5.2.1. Faure. Faure and Mazare [25, 26] have suggested an asynchronous cellular architecture which consists of a 2D mesh of size 65 × 65 for ANN. Each PE has a routing and a processing part running at 20 MHz. The routing is intended to control up to four message transfers in parallel. Using 16 bits for weights and an array configured for NETtalk the designers claim 51.4 MCUPS. Basically, node parallelism is used but each node is distributed over two PEs.

9.5.2.2. Hitachi. Masaki *et al.* and Yasunaga *et al.* at Hitachi have developed a wafer scale integration (WSI) neural network [81, 137, 138]. On one 5-inch silicon wafer they can have 540 neurons, each having up to 64 weights (the 64 largest values are chosen). Node parallelism is used, and the neurons communicate through a time-shared digital bus. Each PE has an 8 by 9-bit multiplier and a 16-bit accumulator. The measured time step is 464 ns. The performance of a wafer is then 138 MCPS. They intend to build a system out of 8 wafers and could then theoretically achieve 1100 MCPS. No on-chip learning is available.

9.5.2.3. SIEMENS. Ramacher and his colleagues at Siemens [102–104] have suggested and built parts of a 2D array composed of systolic neural signal processor modules. The basic components of their MA16 chip are pipelined 16-bit multipliers together with adders. In this respect the design is similar to CNAPS. However, the Siemens architecture does not use broadcast-based communication, but instead uses a systolic data flow (partitioned in groups of four). Each chip has a throughput on the order of 500 MCPS. To get a complete system, 256 MA16 chips are concatenated, which should give a maximum performance of 128 GCPS. No estimated learning rates have been given.

10.0. DISCUSSION AND CONCLUSIONS

10.1. ANN and Parallel Computers

Practically every powerful, parallel computer will do well on ANN computations that use training example parallelism. It is the kind of computation that even brings the performance close to the peak performance. This is of course interesting for people who do research on training algorithms or do application development where the training of the system can be done in batch. However, for training in real time, this form of parallelism cannot be applied; the user is obliged to use node or weight parallelism instead. This places other demands on the architecture, resulting in lower performance figures.

This is clearly illustrated by the various BP implementations made on the Connection Machine: Singer, relying entirely on training example parallelism, achieves 325 MCUPS. Zhang *et al.*, only partly utilizing training example parallelism, reach 40 MCUPS. Rosenberg and Blelloch, finally, who only use other forms of parallelism, end up with a maximum performance of 13 MCUPS. The latter implementation is so heavily communication bound that it does not even matter if bit parallelism is utilized or not!

However, if the architecture meets the communication demands, near peak performance can be reached also in real-time training. The vertical and horizontal highways of the DAP architecture seem to be the key to the good performance reported for this machine [90]. On a computer where the maximum number of 8-bit multiply-and-add operations per second is 450–700M, 160 MCUPS (8 bit) implies a very little amount of communication overhead.

A major conclusion from this survey is that the regularity of ANN computations suits SIMD architectures perfectly; in none of the implementations studied has a real MIMD division of the computational task been required.

The majority of ANN computations following the most popular models of today can be mapped rather efficiently onto existing architectures. However, some of the models, for example, SDM, require a highly or massively parallel computer capable of performing tailored, but simple, operations in parallel and maintaining a very large amount of storage.

10.2. Communication

Broadcast and ring communication can be very efficiently utilized in ANN computations. In highly parallel machines, broadcast or ring communication alone has proved to be sufficient. For massively parallel machines it is difficult to use only broadcast without using training example parallelism. This is due to the fact that the number of nodes in each layer, or the number of inputs to each node, seldom is as large as the number of PEs in the machine. On a two-dimensional mesh machine, broadcast in one direction at a time may be used and node and weight parallelism may be combined. Thus, broadcast is an extremely useful communication facility also in these machines. The "highways" of the DAP architecture serve this purpose.

10.3. Bit-Serial Processing

Bit-serial processor arrays are very promising host machines for ANN computations. Linear arrays, or arrays with broadcast, are suitable for utilizing node parallelism. In mesh-connected arrays node and weight parallelism may be used simultaneously, if desired. Multiplication is the single most important operation in ANN computations. Therefore, there is much to gain in the bit-serial architectures if support for fast multiplication is added, as shown in Centipede and the REMAP^3 project.

As an illustration of this we can compare the performance figures for the implementations of BP on AAP-2 and REMAP^3, respectively. On the 64K PE AAP-2 machine, which lacks support for multiplication, 18 MCUPS using 26 bits data are reported on a network that suits the machine perfectly. The same performance can be achieved on a 512-PE linear array REMAP^3 implementation, in which bit-serial multipliers are used. AAP-2 also lacks a fast broadcasting facility, but this is of minor importance compared to the slow multiply operations.

10.4. Designing a General ANN Computer

A fruitful approach when designing a massively or highly parallel computer for general ANN computations is to start with a careful analysis of the requirements that are set by the low-level arithmetic operations and design processing elements which meet these demands. Then an architecture is chosen that makes it possible to map the computations on the computational structure in a way that makes processing and communication balanced.

It seems that broadcast communication often is a key to success in this respect, since it is a way to time-share communication paths efficiently. The approach has been used in both the CNAPS and the REMAP^3 design processes, both resulting in "only" highly (not massively) parallel modules with broadcast, the former with bit-parallel processors, the latter with bit-serial ones. Neither design utilizes all the available parallelism; instead they leave weight parallelism to be serialized on the same processor. Both reach near peak performance on a variety of algorithms.

10.5. Implementing Artificial Neural Systems

The real challenge for computer architects in connection with the neural network area in the future lies in the implementation of Artificial Neural Systems, i.e., sys-

tems composed of a large number of cooperating modules of neural networks. Each of the modules should be allowed to implement a different network structure, and the modules must be able to interact in different ways and at high speed. This implies that heterogeneous systems composed of homogeneous processing arrays must be developed, and that special attention must be paid to the problem of interaction between modules and between peripheral modules and the environment. The role of MIMD architectures in neural processing probably lies in this area, actually meaning that MIMSIMD (Multiple Instruction streams for Multiple SIMD arrays) architectures will be seen.

These new computer architectures are sometimes referred to as "sixth-generation computers," or "action-oriented systems" [2, 3], since they are capable of interacting with the environment using visual, auditory, or tactile sensors, and advanced motor units.

So far these matters have not been addressed by very many computer architects (nor by artificial neural network researchers). We believe that flexible, massively (or maybe only highly) parallel modules are important tools in experimental work aimed at building such systems for qualified real-time pattern recognition tasks.

ACKNOWLEDGMENT

This work has been partly financed by the Swedish National Board for Technical Development (NUTEK) under contract no. 900-1583.

REFERENCES

1. Almeida, L. D. Backpropagation in perceptrons with feedback. In *NATO ASI Series: Neural Computers*. Neuss, Federal Republic of Germany, 1987.
2. Arbib, M. A. *Metaphorical Brain 2: An Introduction to Schema Theory and Neural Networks*. Wiley–Interscience, New York, 1989.
3. Arbib, M. A. Schemas and neural network for sixth generation computing. *J. Parallel Distrib. Comput.* **6,** 2 (1989), 185–216.
4. Asogawa, M., Nishi M., and Seo, Y. Network learning on the supercomputer. In *Proc. 36th IPCJ Meeting,* 1988, pp. 2321–2322. [In Japanese]
5. Barash, S. C., and Eshera, M. A. The systolic array neurocomputer: Fine-grained parallelism at the synaptic level. In *First International Joint Conference on Neural Networks,* 1989.
6. Beck, J. The ring array processor (RAP): Hardware (Tech. Rep. 90-048, International Computer Science Institute, Berkeley, CA, 1990.
7. Beetem, J., Denneau, M., and Weingarten, D. The GF11 parallel computer. In Dongarra (Ed.). *Experimental Parallel Computing Architectures*. North-Holland, Amsterdam, 1987.
8. Blank, T. The MasPar MP-1 Architecture. In *Proc. COMPCON Spring 90,* San Francisco, CA, 1990, pp. 20–24.
9. Blevins, D. W., *et al.* Blitzen: A highly integrated massively parallel machine. *J. Parallel Distrib. Comput.* (1990), 150–160.
10. Bowler, K. C., *et al. An Introduction to OCCAM 2 Programming*. Studertlitteratur, Lund, Sweden, 1987.
11. Brown, J. R., Garber, M. M. and Vanable, S. F. Artificial neural network on a SIMD architecture. In *Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation*. Fairfax, VA, 1988, pp. 43–47.
12. Brown, N. H., Jr. Neural network implementation approaches for the Connection machine. In *Neural Information Processing Systems,* Denver, CO, 1987, pp. 127–136.
13. Caviglia, D. D., Valle, M., and Bisio, G. M. Effects of weight discretization on the back propagation learning method: Algorithm design and hardware realization. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 631–637.
14. Chinn, G., *et al.* Systolic array implementations of neural nets on the MasPar MP-1 massively parallel processor. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 169–173.
15. Christy, P. Software to support massively parallel computing on the MasPar MP-1. In *Proc. COMPCON Spring 90,* San Fransisco, CA, 1990, pp. 29–33.
16. Cloud, E., and Holsztynski, W. Higher efficiency for parallel processors. In *Proc. IEEE SOUTHCON,* 1984.
17. DARPA. *Neural Network Study*. AFCEA. Fairfax, VA, 1988.
18. Deprit, E. Implementing recurrent back-propagation on the Connection Machine. *Neural Networks* **2,** 4 (1989), 295–314.
19. Diegert, C. Out-of-core backpropagation. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 97–103.
20. Duranton, M., and Sirat, J. A. Learning on VLSI: A general purpose digital neurochip. In *International Conference on Neural Networks,* Washington, DC, 1989.
21. Duranton, M., and Sirat, J. A. Learning on VLSI: A general-purpose digital neurochip. *Philips J. Res.* **45,** 1 (1990), 1–17.
22. Fahlman, S. Benchmarks for ANN. Information and data sets of common benchmarks have been collected by Scott Fahlman and are available by internet ftp. The machine for anonymous ftp are "pt.cs.cmu.edu" (128.2.254.155) below the directory "/afs/cs/project/connect/bench". 1990.
23. Fahlman, S. E. An empirical study of learning speed in back-propagation networks. Rep. No. CMU-CS-88-162, Carnegie Mellon, 1988.
24. Fahlman, S. E., and Lebiere, C. The cascade-correlation learning architecture. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 524–532.
25. Faure, B., and Mazare, G. *Microprocessing and Microprogramming, A Cellular Architecture Dedicated to Neural Net Emulation*. Vol. 30, North-Holland, Amsterdam, 1990.
26. Faure, B., and Mazare, G. Implementation of back-propagation on a VLSI asynchronous cellular architecture. In *International Neural Networks Conference,* Paris, 1990, Vol. 2, pp. 631–634.
27. Fernström, C., Kruzela, I., and Svensson, B. *LUCAS Associative Array Processor—Design, Programming and Application Studies*. Springer-Verlag, Berlin, 1986, *Lecture Notes in Computer Science,* Vol. 216.
28. Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Trans. Comput.* **C-21** (1972), 948–960.
29. Forrest, B. M., *et al.* Implementing neural network models on parallel computers. *Comput. J.* **30,** 5 (1987), 413–419.
30. Forrest, B. M., *et al.* Neural network models. In *International Conference on Vector and Parallel Processors in Computational Science III,* Liverpool, UK, 1988, Vol. 8, pp. 71–83.
31. Frazier, G. TeraOPs and TeraBytes for neural networks research. *Texas Instruments Tech. J.* **7,** 6 (1990), 22–33.
32. Fujimoto, Y. An enhanced parallel planar lattice architecture for

large scale neural network simulations. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 581–586.

33. Fujimoto, Y., and Fukuda, N. An enhanced parallel planar lattice architecture for large scale neural network simulations. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 2, pp. 581–586.
34. Fukuda, N., Fujimoto, Y., and Akabane, T. A transputer implementation of toroidal lattice architecture for parallel neurocomputing. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 43–46.
35. Grajski, K. A. Neurocomputing using the MasPar MP-1. In Przytula and Prasanna (Eds.), *Digital Parallel Implementations of Neural Networks*. Prentice–Hall, Englewood Cliffs, NJ, 1992.
36. Grajski, K. A., *et al.* Neural network simulation on the MasPar MP-1 massively parallel processor. In *The International Neural Network Conference,* Paris, France, 1990.
37. Hammerstrom, D. A VLSI architecture for high-performance, low-cost, on-chip learning. In *International Joint Conference on Neural Networks,* San Diego, 1990, Vol. 2, pp. 537–543.
38. Hammerstrom, D., and Nguyen, N. An implementation of Kohonen's self-organizing map on the Adaptive Solutions neurocomputer. In *International Conference on Artificial Neural Networks,* Helsinki, Finland, 1991, Vol. 1, pp. 715–720.
39. Hecht-Nielsen, R. Theory of the backpropagation neural networks. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 1, pp. 593–605.
40. Hillis, W. D., and Steel, G. L. J. Data parallel algorithms. *Comm. ACM.* **29,** 12 (1986), 1170–1183.
41. Hinton, G. E., and Sejnowski, T. J. Optimal perceptual inference. In *Proc. IEEE Computer Society Conference on Computer Vision & Pattern Recognition,* Washington, DC, 1983, pp. 448–453.
42. Hinton, G. E., and Sejnowski, T. J. Learning and relearning in Boltzmann machines. In Rumelhart and McClelland (Eds.). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* Vol. 2, *Psychological and Biological Models.* MIT Press, Cambridge, MA, 1986.
43. Hiraiwa, A., *et al.* A two level pipeline RISC processor array for ANN. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 137–140.
44. Hockney, R. W., and Jesshope, C. R. *Parallel Computer 2.* Adam Hilger, Bristol, United Kingdom, 1988.
45. Hodges, R. E., Wu, C.-H., and Wang, C.-J. Parallelizing the self-organizing feature map on multi-processor systems. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 141–144.
46. Holler, M. A. VLSI implementations of neural computation models: A review. Internal Report, Intel Corp., Santa Clara, CA, 1991.
47. Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. USA* **79** (1982), 2554–2558.
48. Hopfield, J. J. Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Nat. Acad. Sci. USA* **81** (1984), 3088–3092.
49. Hopfield, J. J., and Tank, D. Computing with neural circuits: A model. *Science* **233** (1986), 624–633.
50. Hubel, D. H. *Eye, Brain and Vision.* Scientific American Library, New York, 1988.
51. Hunt, D. J. AMT DAP—A processor array in a workstation environment. *Comput. Systems Sci. Engrg.* **4,** 2 (1989), 107–114.
52. Hwang, J.-N., Vlontzos, J. A., and Kung, S.-Y. A systolic neural network architecture for hidden markov models. *IEEE Trans. Acoustics Speech Signal Process.* **37,** 12 (1989), 1967–1979.
53. INMOS Limited. *Occam Programming Model.* Prentice–Hall, New York, 1984.
54. INMOS Limited. The Transputer family 1987, 1987.
55. INMOS Limited. *Occam 2 Reference Manual.* Prentice–Hall, London, 1988.
56. Jaeckel, L. A. Some methods of encoding simple visual images for use with a sparse distributed memory, with applications to character recognition. Tech. Rep. 89.29, RIACS, NASA Ames Research Center, Moffet Field, CA, 1989.
57. Kanerva, P. Adjusting to variations in tempo in sequence recognition. *Neural Networks Suppl. INNS Abstracts* **1** (1988), 106.
58. Kanerva, P. *Sparse Distributed Memory.* MIT press, Cambridge, MA, 1988.
59. Kanerva, P. Personal communication, 1990.
60. Kassebaum, J., Tenorio, M. F., and Schaefers, C. The cocktail party problem: Speech/data signal separation comparison between backpropagation and SONN. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 542–549.
61. Kato, H., *et al.* A parallel neurocomputer architecture towards billion connection updates per second. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 47–50.
62. Kohonen, T. *Self-Organization and Associative Memory.* Springer-Verlag. Berlin, 1988, 2nd ed.
63. Kohonen, T. The self-organizing map. Proc. IEEE **78,** 9 (1990), 1464–1480.
64. Koikkalainen, P. MIND: A specification formalism for neural networks. In *International Conference on Artificial Neural Networks,* Helsinki, Finland, 1991, Vol. 1, pp. 579–584.
65. Koikkalainen, P., and Oja, E. Specification and implementation environment for neural networks using communication sequential processes. In *International Conference on Neural Networks,* San Diego, CA, 1988.
66. Koikkalainen, P., and Oja, E. The CARELIA simulator: A development and specification environment for neural networks. Res. Rep. 15/1989, Lappeenranta University of Tech, Finland, 1989.
67. Krikelis, A., and Grözinger, M. Implementing neural networks with the associative string processor. In *International Workshop for Artificial Intelligence and Neural Networks,* Oxford, 1990.
68. Kung, H. T. Why systolic architectures? *IEEE Comput.* (Jan. 1982), 37–46.
69. Kung, H. T. The Warp computer: Architecture, implementation and performance. *IEEE Trans. Comput.* (Dec. 1987).
70. Kung, S. Y. Parallel architectures for artificial neural nets. In *International Conference on Systolic Arrays,* San Diego, CA, 1988, pp. 163–174.
71. Kung, S. Y., and Hwang, J. N. Parallel architectures for artificial neural nets. In *International Conference on Neural Networks,* San Diego, CA, 1988, Vol. 2, pp. 165–172.
72. Kung, S. Y., and Hwang, J. N. A unified systolic architecture for artificial neural networks. *J. Parallel Distrib. Comput.* (Apr. 1989).
73. Lawson, J.-C., Chams, A., and Herault, J. SMART: How to simulate huge networks. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 577–580.
74. Lea, R. M. ASP: A cost effective parallel microcomputer. *IEEE Micro.* (Oct. 1988), 10–29.
75. Linde, A., and Taveniku, M. LUPUS—A reconfigurable prototype for a modular massively parallel SIMD computing system."

Masters Thesis Rep. 1991:028E, University of Luleå, Sweden, 1991. [In Swedish]

76. Lippmann, R. P. An introduction to computing with neural nets. *IEEE Acoustics Speech Signal Process. Mag.* **4** (Apr. 1987), 4–22.
77. MacLennan, B. J. Continuous computation: Taking massive parallelism seriously. In *Los Alamos National Laboratory Center for Nonlinear Studies 9th Annual International Conference, Emergent Computation,* Los Alamos, NM, 1989. [Poster presentation]
78. Makram-Ebeid, S., Sirat, J. A., and Viala, J. R. A rationalized error back-propagation learning algorithm. In *International Joint Conference on Neural Networks,* Washington, DC, 1989.
79. Mann, R., and Haykin, S. A parallel implementation of Kohonen feature maps on the Warp systolic Computer. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 84–87.
80. Marchesi, M., *et al.* Multi-layer perceptrons with discrete weights. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 623–630.
81. Masaki, A., Hirai, Y., and Yamada, M. Neural networks in CMOS: A case study. *Circuits and Devices* (July 1990), 12–17.
82. Morgan, N. The ring array processor (RAP): Algorithms and architecture. Tech. Rep. 90-047), International Computer Science Institute, Berkeley, CA, 1990.
83. Morgan, N., *et al.* The RAP: A ring array processor for layered network calculations. In *Proc. Conference on Application Specific Array Processors,* Princeton, NJ, 1990, pp. 296–308.
84. Murray, A. F. Silicon implementation of neural networks. *IEE Proc. F.* **138,** 1 (1991), 3–12.
85. Murray, A. F., Smith, A. V. W., and Butler, Z. F. Bit-serial neural networks. In *Neural Information Processing Systems,* Denver, CO 1987, pp. 573–583.
86. Nakayama, K., Inomata, S., and Takeuchi, Y. A digital multilayer neural network with limited binary expressions. In *International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 587–592.
87. Nickolls, J. R. The design of the MasPar MP-1: A cost effective massively parallel computer. In *Proc. COMPCON Spring 90,* San Fransisco, CA, 1990, pp. 25–28.
88. Nordström, T. Designing parallel computers for self organizing maps. Res. Rep. TULEA 1991:17), Luleå University of Technology, Sweden, 1991.
89. Nordström, T. Sparse distributed memory simulation on REMAP3. Res. Rep. TULEA 1991:16, Luleå University of Technology, Sweden, 1991.
90. Núñez, F. J., and Fortes, J. A. B. Performance of connectionist learning algorithms on 2-D SIMD processor arrays. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 810–817.
91. Obermayer, K., Ritter, H., and Schulten, K. Large-scale simulations of self-organizing neural networks on parallel computers: Application to biological modelling. *Parallel Comput.* **14,** 3 (1990), 381–404.
92. Parker, K. L. Parallelized back-propagation training and its effectiveness. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 179–182.
93. Penz, P. A. The closeness code: An integer to binary vector transformation suitable for neural network algorithms.'' In *International Conference on Neural Networks,* San Diego, CA, 1987, Vol. 3, pp. 515–522.
94. Pesulima, E. E., Panadya, A. S., and Shankar, R. Digital implementation issues of stochastic neural networks. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 187–190.
95. Peterson, C., and Andersson, J. A mean field theory learning algorithm for neural networks. *Complex Systems* **1** (1987), 995–1019.
96. Peterson, C., and Hartman, E. Explorations of mean field theory learning algorithm. *Neural Networks* **2,** 6 (1989), 475–494.
97. Peterson, C., Sutton, J., and Wiley, P. iWarp: A 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro.* (June 1991), 26–29, 81–87.
98. Petkov, N. Systolic simulation of multilayer, feedforward neural networks. *Parallel Process. Neural Systems Comput.* (1990), 303–306.
99. Pineda, F. J. Generalization of back-propagation to recurrent neural networks. *Phys. Rev. Lett.* **59,** 19 (1987), 2229–2232.
100. Pomerleau, D. A., *et al.* Neural network simulation at warp speed: How we got 17 million connections per second. In *Proc. IEEE International Conference on Neural Networks,* San Diego, CA, 1988.
101. Potter, J. L. *The Massively Parallel Processor.* MIT Press. Cambridge, MA, 1985.
102. Ramacher, U., and Beichter, J. Architecture of a systolic neuroemulator. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 2, pp. 59–63.
103. Ramacher, U., *et al.* Design of a 1st generation neurocomputer. In Ramacher, U., & Rückert, U. (Eds.), *VLSI Design of Neural Networks,* Kluwer-Academic Publishers, Dordrecht, The Netherlands, 1991.
104. Ramacher, U., and Wesseling, M. Systolic synthesis of neural networks. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 572–576.
105. Rogers, D. Kanerva's sparse distributed memory: An associative memory algorithm well-suited to the connection machine. In *Proc. Conference on Scientific Application of the Connection Machine,* Moffet Field, CA, 1988, Vol. 1, pp. 282–298.
106. Rogers, D. Kanerva's sparse distributed memory: An associative memory algorithm well-suited to the connection machine. Tech. Rep. 88.32, RIACS, NASA Ames Research Center, 1988.
107. Rogers, D. Statistical prediction with Kanerva's sparse distributed memory. In *Neural Information Processing Systems 1,* Denver, CO, 1988, pp. 586–593.
108. Rosenberg, C. R., and Blelloch, G. An implementation of network learning on the connection machine. In *Proc. 10th International Conference on AI,* Milan, Italy, 1987, pp. 329–340.
109. Rosenblatt, F. *Principles of Neurodynamics.* Spartan Books, New York, 1959.
110. Rumelhart, D. E., and McClelland, J. L. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, Cambridge, MA, 1986, Vols. I and II.
111. Rumelhart, D. E., and McClelland, J. L. *Explorations in Parallel Distributed Processing.* MIT Press, Cambridge, MA, 1988.
112. Sami, M., and Calzadilla-Daguerre, J. *Silicon Architectures for Neural Nets.* North-Holland, Amsterdam, 1991.
113. Schmitt, R. S., and Wilson, S. S. The AIS-5000 parallel processor. *IEEE Trans. Pattern Anal. Mach. Intell.* **10,** 3 (1988), 320–330.
114. Sejnowski, T. J., and Rosenberg, C. R. Parallel networks that learn to pronounce English. *Complex Systems* **1** (1987) 145–168.
115. Shams, S., and Przytula, K. W. Mapping of neural networks onto programmable parallel machines. In *Proc. IEEE International Symposium on Circuits and Systems,* New Orleans, LA, 1990.
116. Shoemaker, P. A., Carlin, M. J., and Shimabukuro, R. L. Backpropagation with coarse quantization of weight updates. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 1, pp. 573–576.
117. Siemon, H. P., and Ultsch, A. Kohonen networks on transputers:

Implementation and animation. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 643–646.

118. Singer, A. Exploiting the Inherent parallelism of artificial neural networks to achieve 1300 million interconnects per second. In *International Neural Network Conference,* Paris, France, 1990, Vol. 2, pp. 656–660.
119. Singer, A. Implementation of artificial neural networks on the Connection Machine Tech. Rep. RL90-2, Thinking Machine, Corp., 1990.
120. Singer, A. Implementations of artificial neural networks on the Connection Machine. *Parallel Comput.* **14,** 3 (1990), 305–316.
121. Sirat, J. A., *et al.* Unlimited accuracy in layered networks. In *First IEE Conference on Artificial Neural Networks,* London, 1989, pp. 181–185.
122. Stevenson, M., Winter, R., and Widrow, B. Sensitivity of layered neural networks to errors in the weights. In *International Joint Conference on Neural Networks,* Washington, DC, 1990, Vol. 1, pp. 337–340.
123. Stevensson, M., Winter, R., and Widrow, B. Sensitivity of feed-forward neural networks to weight errors. *IEEE Trans. Neural Networks* **1,** 1 (1990), 71–80.
124. Svensson, B., and Nordström, T. Execution of neural network algorithms on an array of bit-serial processors. In *10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition,* Atlantic City, NJ, 1990, Vol. II, pp. 501–505.
125. Tenorio, M. F., and Lee, W.-T. Self-organizing network for optimum supervised learning. *IEEE Trans. Neural Networks* **1** (1990).
126. Tenorio, M. F. d. M. Topology synthesis networks: Self-organization of structure and weight adjustment as a learning paradigm. *Parallel Comput.* **14,** 3 (1990), 363–380.
127. Thinking Machines Corporation. Connection Machine, Model CM-2 Technical Summary. Version 5.1, TMC, Cambridge, MA, 1989.
128. Tomboulian, S. Introduction to a system for implementing neural net connections on SIMD architectures. In *Neural Information Processing Systems,* pp. 804–813, Denver, CO, 1987, pp. 804–813.
129. Treleaven, P., Pacheco, M., and Vellasco, M. VLSI architectures for neural networks. *IEEE Micro.* (Dec. 1989), 8–27.
130. Treleaven, P. C. PYGMALION neural network programming environment. In *International Conference on Artificial Neural Networks,* Helsinki, Finland, 1991, Vol. 1, pp. 569–578.
131. von Neumann, J. *The Computer and the Brain.* Yale Univ. Press, New Haven, CT, 1958.
132. Watanabe, T., *et al.* Neural network simulation on a massively parallel cellular array processor: AAP-2. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 2, pp. 155–161.
133. Whitby-Stevens, C. Transputers—Past, present, and future." *IEEE Micro.* (Dec. 1990), 16–82.
134. Willshaw, D. J., Buneman, O. P., and Longuet-Higgins, H. C. Non-holographic associative memory. *Nature* **222** (1969), 960–962.
135. Wilson, S. S. Neural computing on a one dimensional SIMD array. In *11th International Joint Conference on Artificial Intelligence,* Detroit, MI, 1989, pp. 206–211.
136. Witbrock, M., and Zagha, M. An implementation of back-propagation learning on GF11, a large SIMD parallel computer. Rep. No. CMU-CS-89-208, Computer Science, Carnegie Mellon, 1989.
137. Yasunaga, M., *et al.* A wafer scale integration neural network utilizing completely digital circuits. In *International Joint Conference on Neural Networks,* Washington, DC, 1989, Vol. 2, pp. 213–217.
138. Yasunaga, M., *et al.* Design, fabrication and evaluation of a 5-inch wafer scale neural network LSI composed of 576 digital neurons. In *Proc. International Joint Conference on Neural Networks,* San Diego, CA, 1990, Vol. 2, pp. 527–535.
139. Zhang, X., *et al.* An efficient implementation of the backpropagation algorithm on the Connection Machine CM-2. In *Neural Information Processing Systems 2,* Denver, CO, 1989, pp. 801–809.
140. Åhlander, A. and Svensson, B. Floating-point calculations in bit-serial *SIMD* computers. Res. Rep. CDv-9104, Halmstad University, Sweden, 1991.

Received August 19, 1991; revised October 1, 1991; accepted October 18, 1991

TOMAS NORDSTRÖM was born May 19, 1963, in Härnösand, Sweden. He holds an M.S. degree in computer science and engineering from Luleå University of Technology, Sweden. Since 1988 he has been a Ph.D. student at the same University, and since 1991 he has held a Licentiate degree in computer engineering. His research interests include parallel architectures and artificial neural networks.

BERTIL SVENSSON was born February 15, 1948, in Eldsberga, Sweden. He received his M.S. degree in electrical engineering in 1970 and his Ph.D. degree in computer engineering in 1983, both from the University of Lund, Sweden, where he also served as a researcher and assistant professor. In 1983 he joined Halmstad University College, Sweden, as assistant professor and became vice president the same year. Until recently he was a professor in computer engineering at Luleå University of Technology, Sweden, which he joined in 1989. Presently he is a professor in computer engineering at Chalmers University of Technology, Sweden. He is also head of the Centre for Computer Science, a research institution of Halmstad University. He has been working in the field of highly parallel SIMD computers and their applications, e.g., image processing, since 1978. Currently he is conducting research on the design of modular, reconfigurable, massively parallel computers for trainable, embedded systems.

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.

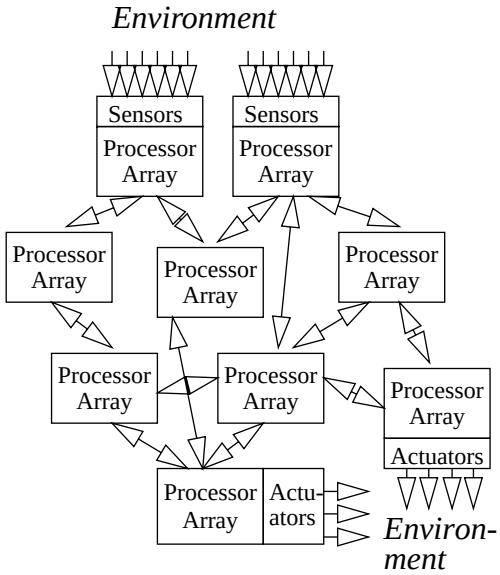

Figure 1. A multi-module architecture for an action-oriented system

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.

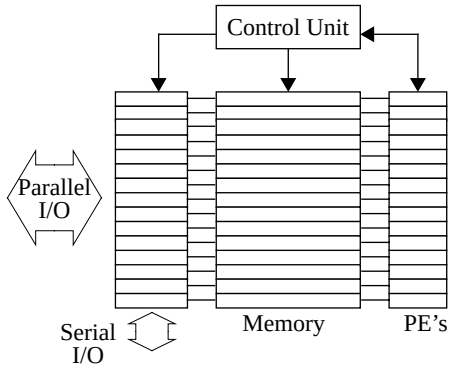

Figure 2. SIMD Module

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.

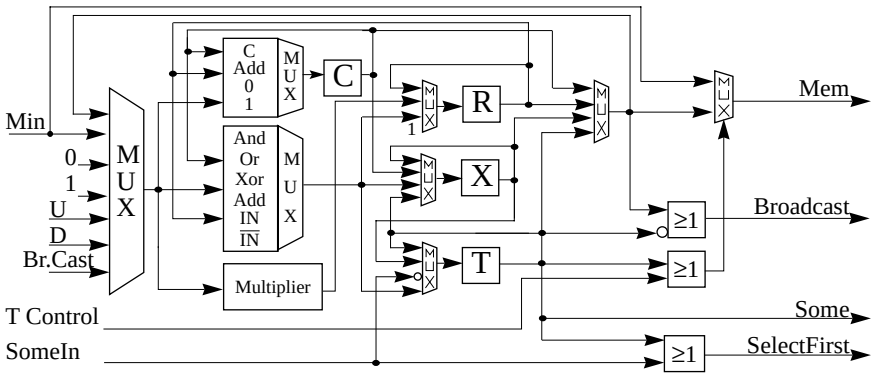

Figure 3. Sample PE from REMAP^3

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).

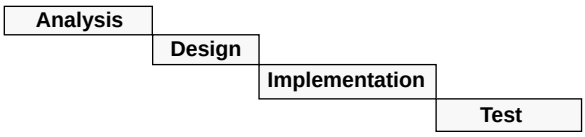

Figure 4. The waterfall model.

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.

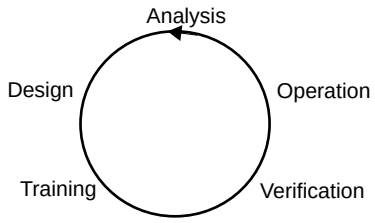

Figure 5. The circular development model.

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*.

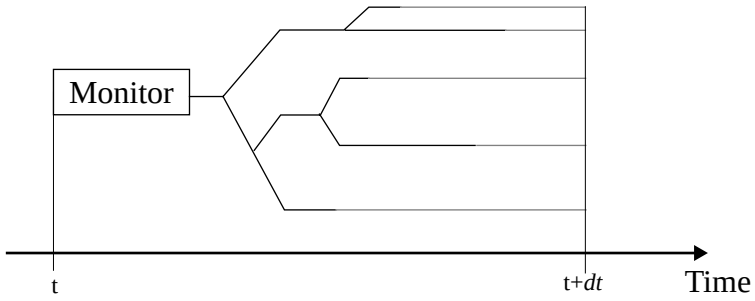

Figure 6. Temporal view of Monitor and Work Process

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.

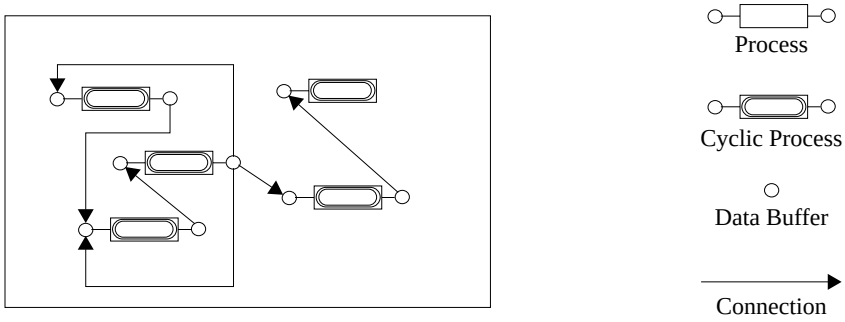

Figure 7. System level display (left) and basic symbols.

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.

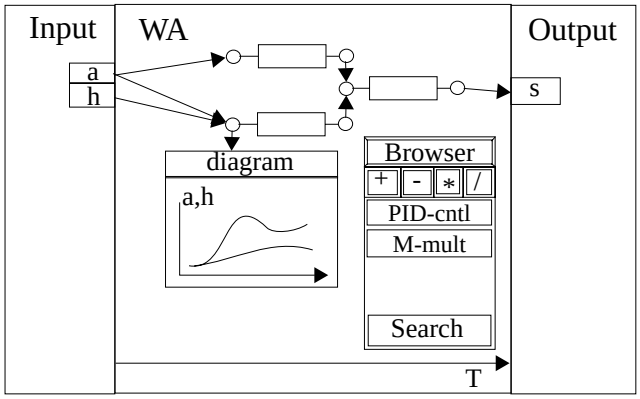

Figure 8. Node level (or lower levels) display

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.

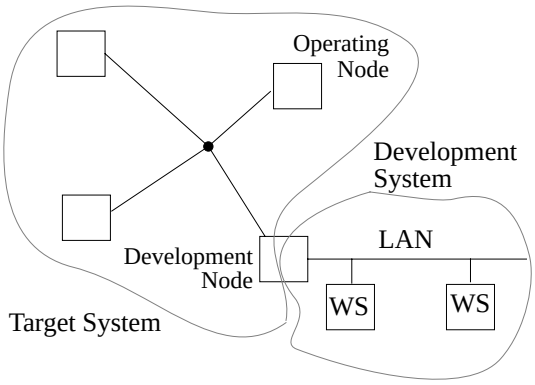

Figure 9. Multi-node target system and multiple-workstation development system

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".

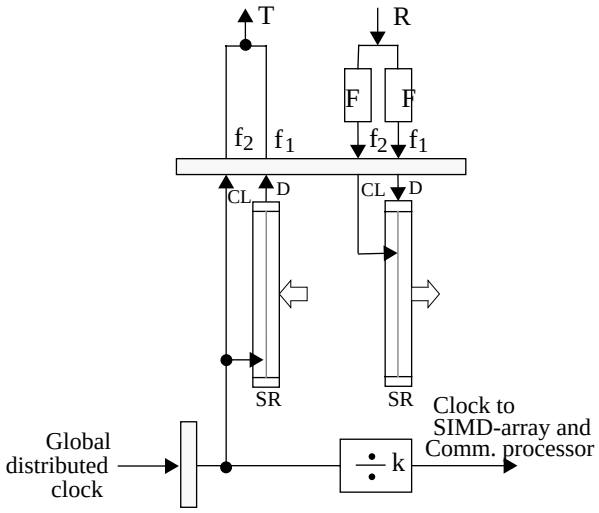

Figure 10. Communication interface. T is transmit, R is receive. Grey boxes indicate the optical/ electrical conversion.

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 exe-

cutes the statically scheduled send and receive commands necessary for carrying out the communication pattern of the node (Figure 11).

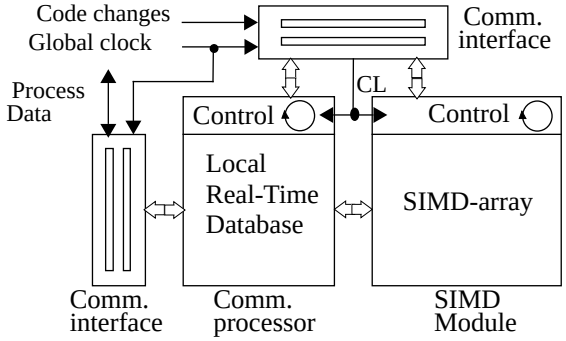

Figure 11. A Node

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.

Lecture Notes in Computer Science 705, Herbert Grünbacher, Reiner W. Hartenstein Eds. Springer-Verlag
Field-Programmable Gate Array: Architectures and Tools for Rapid Prototyping; *Selected papers from: Second International Workshop on Field-Programmable Logic and Applications (FPL'92), Vienna, Austria*
pp.199-210

Using FPGAs to Implement a Reconfigurable Highly Parallel Computer

Arne Linde*, Tomas Nordström‡, and Mikael Taveniku*

* Department of Computer Engineering
Chalmers University of Technology,
S-41296 Göteborg, Sweden

‡ Division of Computer Engineering
Luleå University of Technology,
S-95187 Luleå, Sweden

E-mail: arne@ce.chalmers.se, tono@sm.luth.se, micke@ce.chalmers.se

Abstract. With the arrival of large Field Programmable Gate Arrays (FPGAs) it is possible to build an entire computer using only FPGA and memory. In this paper we share some experience from building a highly parallel computer using this concept. Even if today's FPGAs are of considerable size, each processor must be relatively simple if a highly parallel computer is to be constructed from them. Based on our experience of other parallel computers and thorough studies of the intended applications, we think it is possible to build very powerful and efficient computers using bit-serial processing elements with SIMD (Single Instruction stream, Multiple Data streams) control.
A major benefit of using FPGAs is the fact that different architectural variations can easily be tested and evaluated on real applications. In the primary application area, which is artificial neural networks, the gains of extensions like bit-serial multipliers or counters can quickly be found. A concrete implementation of a processor array, using Xilinx FPGAs, is described in this paper.
To get efficient usage and high performance with the FPGA circuits signal flow plays an important role. As the current implementation of the Xilinx EDA software does not support that design issue, the signal flow design has to be made by hand. The processing elements are simple and regular which makes it easy to implement them with the XACT Editor. This gives high performance, up to 40–50 MHz.

1 Introduction

The requirements for flexibility and adaptivity to different circumstances and environments have motivated research and development towards trainable systems rather than programmed ones. This is true especially for “action oriented systems” which interact with their environments by means of sophisticated sensors and actuators, often with a high degree of parallelism [2]. Response time requirements and the demand to accomplish the training task points to highly or massively parallel computer architectures.

In REMAP, the Real-Time, Embedded, Modular, Action-oriented, Parallel Processor Project [3], the potential of distributed SIMD (Single Instruction stream, Multiple Data streams) modules for realization of trainable systems is investigated. Each SIMD

module is a highly parallel computer with simple PEs tuned to efficiently compute artificial neural network algorithms.

Within the project, a series of studies have been performed [10–12, 16] concerning the execution of neural network algorithms on highly parallel SIMD computers, with special emphasis on architectures based on bit-serial processing elements (PEs). The results show that SIMD is the best suited parallel processing paradigm for artificial neural networks (ANNs) and that arrays of bit-serial PEs with simple inter-PE communication are surprisingly efficient. As multiplication is found to be the single most important operation in these computations, there is much to be gained in the bit-serial architecture if support for fast multiplication is added.

Using today's relatively large field-programmable gate arrays (FPGAs), it is possible to build an entire computer using only FPGAs and memory. Still, if a highly parallel computer is to be constructed out of them, each processor must be very simple. As shown in our studies of parallel computers for ANN, bit-serial PEs with SIMD control suit our computational needs, which makes it feasible to use FPGAs as a means to construct the first prototypes of our computers.

The computer built should not be seen as a final "product", it is more of an architecture laboratory, in which it is possible to change the architecture of each PE rapidly. Designing and compiling a new architecture takes about one week and downloading an already prepared architecture takes less than a second.

2 Applications

To realize action-oriented systems, the artificial neural network (ANN) models [6, 7] form a very important implementation class. As shown in [12] the demands on the architecture are quite moderate for standard ANN algorithms like feed-forward networks with back propagation, Hopfield networks, or Kohonen self-organizing maps. These models, like most of the ANN algorithms, use a very simple model of the neuron. Typically, an artificial neuron computes a weighted sum of its inputs, a nonlinear function is then usually applied to the sum, and the result is sent along to neighboring neurons, see Fig. 1. The power of ANN computations comes from the large number of neurons (nodes) and their rich interconnections via synapses (weights).

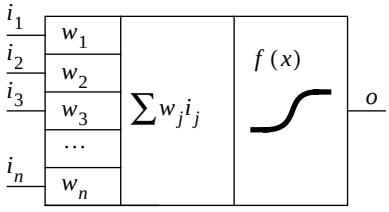

Fig. 1. The simplest model of a neuron. The neuron calculates the weighted sum of its inputs and applies a non-linear function to it, $o = f(\sum w_j i_j)$.

Different ANN models are characterized not only by the type of nodes, but also by the interconnection topology, and the training algorithm used [9]. Common topologies

are layered feed-forward networks, winner take all networks, and all-to-all (Hopfield) networks. Common training rules are error back-propagation and self-organizing feature maps.

Parallelism can be found in many different places [12] but for action-oriented systems the parallelism in the nodes and weights are the important ones (node and weight parallelism). As we are focusing on the ANN models in which one can count the number of nodes and weights in thousands, we will have a lot of parallelism available. These two types of parallelism also fit the SIMD concepts perfectly.

The calculation of the weighted sum is the most time consuming calculation and should therefore be supported architecturally by any computer intended for real-time ANN computations. Also the communication means between different ANN algorithms/modules as well as between these modules and the environment have to be carefully designed.

Another possible application area for the architecture we describe would be low-level image processing. As the architecture is not very different from architectures which are known to perform well on low-level image processing problems (e.g. AIS-5000 [14], LUCAS [5]), this problem area also fits our architecture well.

3 The REMAP Computer

REMAP is an experimental project. A sequence of gradually evolved prototypes are being built, starting with a small, software configurable PE array module, implemented as a Master's thesis project [8]. With only slight modifications in the PE array architecture, but using a new high-performance control unit, the second prototype has now been built[1]. This prototype is almost full-scale with respect to the number of PEs, but far from miniaturized enough for embedded systems. It is the architecture of this prototype that is described in this paper.

The computer consists of a number of computing modules controlled by a master computer. Each computing module is a SIMD computer of its own. It contains a linear array of bit-serial processing elements with memory and I/O-circuits controlled by a control unit, see Fig. 2.

1. A 128 PE prototype has now (beginning of 1993) been completed.

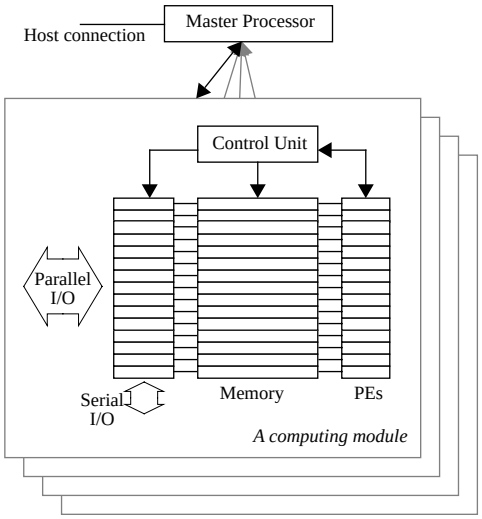

Fig. 2. Overview of the REMAP system. The PEs are implemented in Xilinx XC4005 circuits (8 in each) and the serial/parallel I/O device in Xilinx XC3020 (8 parallel and 8 serial I/O each)

3.1 The Control Unit

The main task for the control unit is to send instructions together with PE memory addresses to the PE array. At the same time it computes new address values (typically increments and decrements).

The control unit currently in use [3] has been designed around a microprogrammable sequencer and a 32 bit ALU (AMD 28331, 28332). The control unit is capable of sending out a new address together with a new instruction every 100 ns. The controller is more general purpose than usually needed, but until we know what is needed it serves our purpose. The microprograms to be executed by the control unit are stored in an 8K words control store. The operations can either be simple field operations, like adding two fields, or whole algorithms like an ANN computation. For the moment only a micro-code assembler is available to program the control unit, but we intend to develop more high level software development tools in the future. Currently we are looking into the possibility of using/developing a data-parallel language similar to C* [17].

3.2 PEs for ANN Algorithms

The detailed studies of artificial neural network computations have resulted in a proposal for a PE that is well suited for this area. The design is depicted in Fig. 3. Important features are the bit-serial multiplier and the broadcast connection. Notably, no other inter-PE connections than broadcast and nearest neighbor are needed. The PE is quite general purpose, and we are confident that this is a useful PE design also in several other application areas. In this version it consists of four flipflops (R, C, T and X),

eight multiplexers, some logic and a multiplication unit. The units get their control signals directly from the micro instruction word sent from the control unit.

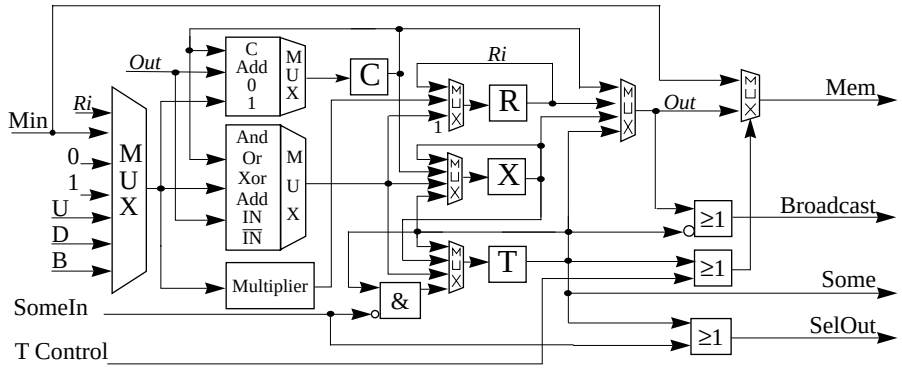

Fig. 3. The sample PE

In simple PEs without support for multiplication the multiplication time grows quadratically with the data length. A method based on carry-save adders [5] (see Fig. 4) can reduce the multiplication time required to the time to load the operands and store the result.

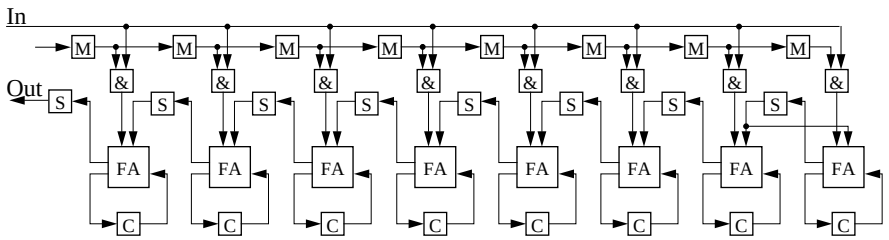

Fig. 4. Design of a two's-complement bit-serial multiplier. It is operated by first shifting in the multiplicand, most significant bit first, into the array of M flip-flops. The bits of the multiplier are then successively applied to the input, least significant bit first. The product bits appear at the output with least significant bit first.

As shown in [11] the incorporation of a counter instead of a multiplier in the PE design may pay off well when implementing the Sparse Distributed Memory (SDM) neural network model. A 256 PE REMAP realization with counters is found to run SDM at speeds 10–30 times that of an 8K PE Connection Machine CM-2, (with frequencies normalized and on an 8K problem). Already without counters (then the PEs become extremely simple) a 256 PE REMAP outperforms a 32 times larger CM-2 by a factor of 4–10. Even if this speed-up for REMAP can be partly explained by the more advanced sequencer, the possibility to tune the PEs for this application is equally important.

3.3 PE Communication

The processing element has two ways of communicating with other processing elements: nearest neighbor and broadcast communication. The nearest neighbor communication network allows each PE to read its neighbor's memory i.e. PE(n) can read from PE(n+1) and PE(n-1). The first and the last PEs are considered neighbors. At any time one of the PEs can broadcast a value to all other PEs or to the control unit. The control unit can also broadcast a value to the PEs. It has also a possibility to check if any of the PEs has the activity bit (T-flip-flop) set. If several PEs are active at the same time and the control unit wants one PE to broadcast, the control unit simply does a select-first operation, which selects the first active PE and deselects the rest. These communication and arbitration operations can be used to efficiently perform matrix computations as well as search and test operations sufficient for many application areas, especially artificial neural networks. To be useful in real-time applications which include interacting with a changing environment, high demands are put on the I/O-system. To meet these demands the processor array is equipped with two I/O-channels, one for 8-bit wide communication and the other for array-wide communications. This interface has a capability to run at speeds up to 80 MHz (burst) which, for a 256 PE array, implies a maximum transfer rate of 20 Gbit/s. Due to limitations in the control unit the I/O-interface currently runs at 10 MHz which reduces the transfer rate to 2.5 Gbit/s.

4 Designing with FPGA Circuits

After a market survey we found that FPGAs from Xilinx [22] would serve our needs best. The structure of the Xilinx circuits is shown in Fig. 5. The chip consists of a number of combinatorial logic blocks (CLB), some input-output blocks (IOB) and an interconnection network (ICN). These circuits are user programmable, thus enabling the CLB, IOB and ICN to be programmed by the user. The configuration of the on-chip configuration RAM is carried out at power up or by a reprogramming sequence. The RAM can be loaded from an external memory or from a microprocessor, the latter is used for REMAP. It takes about 400 ms to reprogram the circuits, thus enabling the master-processor to change the architecture of the processing elements dynamically during the execution of programs.

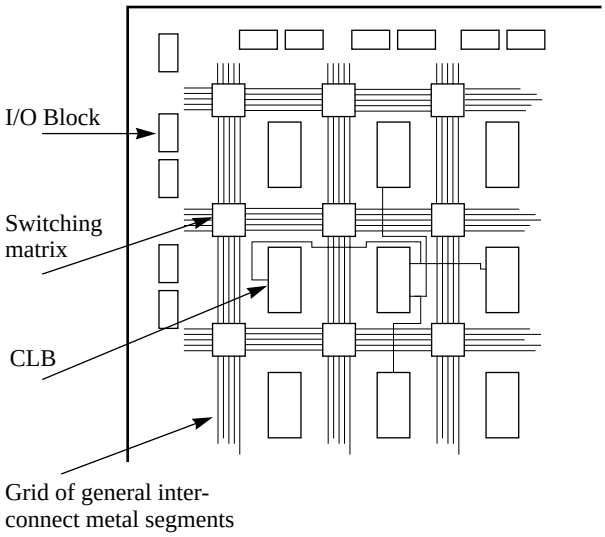

Fig. 5. Xilinx FPGA overview. The IOB connects the I/O-pads to the ICN. These blocks can be configured as input, output or bidirectional blocks. The CLBs are configurable logic blocks consisting of two 16bit (and one 8bit) look-up table for logic functions and two flipflops for state storage. These blocks are only connected to the ICN. The ICN connects the different blocks in the chip. It consist of four kinds of connections: short-range connections between neighboring blocks, medium-range connections connecting blocks on slightly larger distances, long-lines connecting whole rows and columns, and global nets for clock and reset signals broadcasted throughout the whole chip.

Since one of our goals is to make a kind of hardware simulator for different types of PE-architectures using a fixed hardware surrounding, it is required that the connections off chip like those to the memory and control unit have the same function regardless of the currently loaded processor architecture. As shown in [18] it is advantageous to lock the pads so that control signals enter from the top and bottom of the chip, and also design the processing elements so that they are laid out rowwise in the array of CLBs. It is likewise preferable to have a dataflow from left to right in the chip i.e. input data enters the left side and output emerges from the right side.

4.1 Using XC3090

The first prototype was constructed using Xilinx XC3090, and some frustrating experiences were gained from the poor development tools for these circuits. The processing elements in this version are only capable of running at 5 MHz clock frequency. The low speed is due to the incapability of the EDA software to handle signal flow layout in the circuits, something which also leads to low utilization. The PEs were designed using the OrCAD CAE-tools, enabling the designer to work with ordinary logic blocks like multiplexers and different types of gates. The schematics are then automatically converted to suit the Xilinx circuits. This is a fast design method but different parts of the logic become intermixed and long delays are introduced.

4.2 Using XC4005

The current prototype is based on the XC4000 FPGA family from Xilinx. These circuits have a more balanced performance than the XC3000 circuits which have small routing resources compared to the number of CLBs. In the XC4000 family the CLBs are larger, the ICN much more powerful and the internal delays shorter. The circuits range from XC4003, which has a 10 by 10 CLB matrix, to XC4010 with a 20 by 20 CLB matrix, and even larger circuits are announced. With these circuits it is easier to test new types of PEs, as there is more space in them. It will also be possible to increase the maximum clock frequency to 20 MHz, and possibly even 40 MHz if more pipelining is introduced. The greatest advantage with these new circuits is the software; routing a XC3090 chip can take a couple of days on a 80486 machine, while the same problem can be solved in half an hour with the new software for the XC4000 circuits.

One PE of the kind depicted in Fig. 3 occupies approximately 10 CLBs and the eight-bit deep bit-serial multiplier 11 CLBs. Using a XC4005 with a 14 by 14 CLB-matrix, we can get at least 8 PEs in each Xilinx chip. Considering this and the timing demands of 10 MHz operation (due to the control unit), we can easily make design variations both in the main processor and in the multiplier (or other coprocessor). It takes about one week to make a tested and simulated prototype with the XACT editor. The design is of course also open for changes to PEs with other data widths between 2 bits and eight bits.

Tools

High level tools were not available when we started to develop a processing element for the XC4005-circuits. Therefore we have not yet tested how well those tools work. There are, however, several advantages of using the low-level XACT editor in early stages of the design. We get good knowledge of the circuit's limitations and possibilities, and at the same time we get full control of all necessary timing. The usage of XACT is simplified by the regular and simple structure of our design. In the first implementation we aimed towards eight processing elements running at 10 MHz in each XC4005 circuit, based on the previous experiences with the XC3090 circuits. These goals were easily achieved; the eight processing elements can run at 20 MHz utilizing 75% of the XC4005 configurable logic blocks and all of its I/O blocks, this in the 84 pin PLCC package.

Data and Control Flow in the Circuits

The data and control flow play an important role in getting the best performance out of the circuits, therefore we have a basic template with some of the control and data signals already laid out. This template enables the user to easily implement new types of processing elements with minimum effort and at the same time achieve high performance.

When designing the control flow we want to use the global networks as much as possible. This is achieved by using 4 of the global nets and 20 of the vertical long-lines. The memory input signal is connected to the memory output via a horizontal long-line through the chip in order to enable a good data input distribution and allow

write-back of unchanged data when the processing element is inactive. With these restrictions in signal flow the internal delays can be held very low.

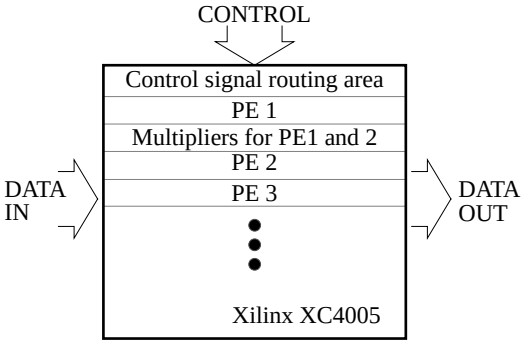

Fig. 6. Layout for PEs in an XC4005

As we have used the XC4005-PC84 which has a 14x14 CLB matrix, and the rest of the hardware is designed for eight processing elements, the chip is divided into four blocks of two processing elements each, occupying three rows in the matrix. Each processing element then gets 21 CLBs, 2 IOBs with PADs, and six IOBs with only edge decoders[1]. After this we have 28 unused CLBs.

Testability

From our experiences of the XC3090 circuits, which sometimes got into undefined states when we tried to reconfigure them, we now separate programming pins to the master processor so that we can directly see which circuit is failing. We also use the possibility of reading back configuration and state data from the Xilinx circuits, which can be done while the PEs are running. The master can also single-step the processor via the control unit and read back all state variables. Two pads on each processing element are dedicated to probing, here we can measure any internal delay simply by loading a configuration with the probe outputs properly programmed (this is done automatically by the XACT EDA software). The JTAG facilities of the XC4000 have not been used, because the PEs, simple as they are, only require a couple of hundred stimuli to excite all modes in them.

The full-scale prototype (256 PEs) can run in 10 MHz with very comfortable timing margins. More memory and additional communication networks can easily be added if need arises.

1. Some of the I/O-blocks in the XC4005-PC84 have no connections to pads. However, these blocks can be used to get a connection to their edge decoder.

5 Other Usage of FPGAs to Run ANN Algorithms

There are other FPGA implementations of ANN models besides ours. A short description of some of them are given below.

A group at North Carolina State University has developed a PC-card called Anyboard [19], which in principle only contains Xilinx chips (4 XC3020s) and RAM. It is part of a "rapid prototyping" environment, where user-specified digital designs can quickly be implemented and tested. One early project using this card was the implementation of a stochastic ANN model called TInMANN [20]. A quite fast and dense implementation was obtained. They used a special purpose architecture, tuned to their algorithm.

Another project, using 25 XC3020s to implement a stochastic Boltzmann machine ANN, was carried out by Skubiszewski [15]. In this implementation the architecture was more like ours (identifiable PEs similar to conventional bit-serial PEs), but no support for the multiplications was included.

Cox and Blanz [4] built an ANN simulator with impressive performance, out of 28 XC3090s. In contrast to the two implementations above and our implementation, they have used a highly specialized bit-parallel approach, which implements a feed-forward neural network of a fixed size (12x14x4).

Another, more specialized, use of FPGAs for ANN computations is made by a group at Tampere University of Technology, Finland [13]. In this group's hardware implementation of Kanerva's Sparse Distributed Memory (SDM), FPGAs are used to implement the main controller as well as more specialized computations like an adder tree. The architecture is highly specialized for SDM and no identifiable processing elements exist.

Xilinx circuits are also used in general hardware emulators such as the Quickturn RPM emulator [21], which emulates designs with from 10K up to 1M gates at a speed of 1MHz. This type of emulators could of course be used to simulate all the designs described in this paper, but with drastically lower speed and CLB utilization.

6 Conclusions and Future Directions

With the REMAP computer, we have a platform from where we can test and evaluate different types of interconnection networks, PE complexities and architectures. This is not restricted to simple bit-serial PEs as the one described in this text, also complex ones such as bit-serial floating point arithmetic units and up to eight bit wide PEs can be implemented. Floating point arithmetic for this platform has been examined by members of the group [1], and will be included. When we have found a good PE architecture we will transfer it to silicon, this decreases the size and increases the system speed. Our aim is to get 256 processing elements, with floating point arithmetic, on each chip running at an internal speed of 200–300MHz.

A robot arm with 12 motors and a number of sensors all controlled in parallel from the array-parallel interface on the REMAP computer is being developed at the Centre for Computer Architecture, Halmstad University. A CCD camera is also planned to be

connected to the byte-wide interface on REMAP as a further step towards a real-time action-oriented system.

To speed up the development cycle in the future some sort of high-level description of the PEs and their interconnections would be needed. From this description it should be possible to generate FPGA layout, VLSI layout, a PE array simulator, and a high level language compiler back-end. Both text-based and graphics based high-level descriptions are considered.

While, in this design of a hardware simulator, we are more interested in the possibilities of changing the processor architecture than to get maximum performance, we have added (retained) the feature that design changes can be made during execution. For example in some parts of an application we may need a counter instead of a multiplier. It is easily accomplished, via program control, to stop the control unit during approximately 400 ms and reprogram the Xilinx circuits.

7 References

1. Åhlander, A. and B. Svensson. "Floating point calculations in bit-serial SIMD computers." In *Fourth Swedish Workshop on Computer Systems Architecture*, Linköping, Sweden, 1992.
2. Arbib, M. A. "Schemas and neural network for sixth generation computing." *Journal of Parallel and Distributed Computing*. Vol. 6(2): pp. 185-216, 1989.
3. Bengtsson, L., A. Linde, T. Nordström, B. Svensson, M. Taveniku and A. Åhlander. "Design and implementation of the REMAP^3 software reconfigurable SIMD parallel computer." In *Fourth Swedish Workshop on Computer Systems Architecture*, Linköping, Sweden, 1992.
4. Cox, C. E. and W. E. Blanz. "GANGLION — A fast field programmable gate array implementation of a connectionist classfier." (RJ 8290 /75651/), IBM Research Division, Almaden Research Centre, 1990.
5. Fernström, C., I. Kruzela and B. Svensson. *LUCAS Associative Array Processor - Design, Programming and Application Studies*. Vol 216 of *Lecture Notes in Computer Science*. Springer Verlag. Berlin. 1986.
6. Hertz, J., A. Krogh and R. G. Palmer. *Introduction to the Theory of Neural Computations*. Addison Wesley. Redwood City, CA. 1991.
7. Kohonen, T. "An introduction to neural computing." *Neural Networks*. Vol. 1: pp. 3-16, 1988.
8. Linde, A. and M. Taveniku. "LUPUS — a reconfigurable prototype for a modular massively parallel SIMD computing system." (Masters Thesis 1991:028 E), University of Luleå, Sweden, 1991. [In Swedish]
9. Lippmann, R. P. "An Introduction to Computing with Neural Nets." *IEEE Acoustics, Speech, and Signal Processing Magazine*. Vol. 4(April): pp. 4-22, 1987.
10. Nordström, T. "Designing parallel computers for self organizing maps." (Res. Rep. TULEA 1991:17), Luleå University of Technology, Sweden, 1991.
11. Nordström, T. "Sparse distributed memory simulation on REMAP3." (Res. Rep. TULEA 1991:16), Luleå University of Technology, Sweden, 1991.

12. Nordström, T. and B. Svensson. "Using and designing massively parallel computers for artificial neural networks." *Journal of Parallel and Distributed Computing*. Vol. 14(3): pp. 260-285, 1992.

13. Saarinen, J., M. Lindell, P. Kotilainen, J. Tomberg, P. Kanerva and K. Kaski. "Highly parallel hardware implementation of sparse distributed memory." In *International Conference on Artificial Neural Networks*, Vol. 1, pp. 673-678, Helsinki, Finland, 1991.

14. Schmitt, R. S. and S. S. Wilson. "The AIS-5000 parallel processor." *IEEE Transaction on Pattern Analysis and Machine Intelligence*. Vol. 10(3): pp. 320-330, 1988.

15. Skubiszewski, M. "A hardware emulator for binary neural networks." In *International Neural Network Conference*, Vol. 2, pp. 555-558, Paris, 1990.

16. Svensson, B. and T. Nordström. "Execution of neural network algorithms on an array of bit-serial processors." In *10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*, Vol. II, pp. 501-505, Atlantic City, New Jersey, USA, 1990.

17. Thinking Machines Corporation. "C* User's guide and C* Programming Guide." (Version 6.0), T M C Cambridge, Massachusetts, 1990.

18. Unnebäck, M. "Gate array implementations of processing elements for a reconfigurable, modular, massively parallel SIMD computer." (Masters Thesis 1991:117 E), Luleå University of Technology, 1991. [In Swedish]

19. Van den Bout, D. E., J. N. Morris, D. Thomae, S. Labrozzi, S. Wingo and D. Hallman. "AnyBoard: An FPGA-based, reconfigurable system." *IEEE Design & Test of Computers*. (September): pp. 21-30, 1992.

20. Van den Bout, D. E., W. Snyder and T. K. Miller III. "Rapid prototyping for neural networks." *Advanced Neural Computers*. Eckmiller ed. North-Holland. Amsterdam. 1990.

21. Wolff, H. "How Quickturn is filling a gap." *Electronics*. (April): 1990.

22. XILINX. *The Programmable Gate Array Data Book*. 1990.

On-Line Localized Learning Systems Part I – Model Description

Tomas Nordström

Division of Computer Science & Engineering
Luleå University of Technology, Sweden
E-mail: tono@sm.luth.se

ABSTRACT

The concept of localized learning systems (LLSs) is introduced in this paper. This concept makes it possible to combine many commonly used artificial neural network (ANN) models into a single "superclass". The LLS model is a feedforward network using an expanded representation with more nodes in the hidden layer than in the input or output layers. The main characteristics of the model are local activity with respect to input space, and localized learning in active nodes.

The principal structure is the same for all the included ANN models whereas they differ in how locality is defined (that is, using different distance measures, receptive field forms, and kernel functions) and in the methods used to train the free parameters. Different ways to vary the LLS concept is studied in detail in this paper. We will however restrict ourselves to methods that can be used in an on-line learning situation.

The connection between a number of well known ANN models and the possible variations of the LLS model is demonstrated. Additionally, these connections let us suggest new variants of "old" ANN models.

The LLS model (that is, the ANN models which constitute this superclass) has been shown to perform classification and approximation tasks very well in comparison to other non LLS models like, for example, multilayer perceptrons trained with error back-propagation, while needing only a fraction of its training time. The LLS model can also easily be shown to be suitable for implementation on parallel computers, a possibility which will be further explored in a companion paper.

1. Introduction

This is the first paper in a series of papers investigating a new class of neural network algorithms which we will refer to as localized learning systems (LLSs). By the concept of LLSs we have extracted a "superclass" of powerful artificial neural network (ANN) models which have so much in common that it seems worthwhile to study a common hardware platform for them. In this first paper we will characterize the LLS model and the ANN it contains. In a companion paper [51] implementations of these models on parallel computers will be discussed. The LLS concept also allows us to find connections between a number of well known ANN models and the possible variations of LLS model. Furthermore, these connections let us suggest new variants of "old" ANN models.

The idea of localized learning (and response) can in some respects be found in Hebb's work [25], dated 1949. Many others have used this idea in the first wave of ANN, i.e., up to the early seventies, concepts like "winner-take-all" [68], potential functions [5, 15], BOXES [44], cerebellar model arithmetic computer (CMAC) [2, 3, 4], competitive learning and self-organization [37, 81], were used to explore the ideas of localized learning. The development of radial basis functions (RBFs) [61] and the revival of the ANN field during the last ten years have resulted in a number of new models containing the idea of localized learning, many built around the concept of RBF networks [8, 46, 47, 58, 59]. This recent interest in ANNs contained in the LLS class has resulted in many important theoretical results. For instance, the universal approximation property shown for the commonly used multilayer perceptrons (MLPs) [12, 17, 28] has also been proven [54] for RBFs and its generalizations. This property implies that any continuous function can be approximated to a given degree of accuracy by a sufficiently large network. One important feature found in all the LLS models is the parallelism in the many nodes used, all using local operations, making these systems eligible for parallel computer implementations.

In earlier studies we have analyzed how to implement ANNs efficiently on parallel computers [49, 50, 52, 77, 78]. In this and the companion paper we continue these studies by analyzing the LLS model.

In the next section we will define the LLS and in the following section we will outline a theoretical foundation for LLSs. This will primarily be based on generalized radial basis functions (GRBFs) [59, 61], one of the most general ANN models in the LLS group. In Section 4 – 6 the feedforward and learning phases are analyzed. In Section 7 we show the similarities and differences between two different LLS variations, the GRBF network and the sparse distributed memory (SDM) model [34, 36]. (Other ANN models are discussed in Appendix A). In this section we also discuss the similarities and differences to the much used MLP trained with back propagation (BP) which is not an LLS model. We end with conclusions and future directions to our research.

2. Localized learning systems (LLSs)

Using the ANN attributes suggested by Lippmann [41]: *network topology*, *node characteristics*, and *learning rules* we define an LLS as:

- *Network topology*: An LLS is a *feedforward artificial neural network*, with only one hidden layer, cf. Figure 1A. The model forms an expanded representation (ER) [76]—the first weight layer performs a nonlinear expansion of the input space into a high dimensional feature space, residing in the hidden layer. Cover shows in [11] that a nonlinear mapping from low-dimensional to high-dimensional space can transform a nonlinear separation problem to a linear one. This fact contributes to the usefulness of ER. The output weight layer is usually linear, and can therefore often be adapted without the local minima problem. The structure is similar to a table lookup where the input is looking up the output value associated with the current input region.
- *Node characteristics*: In an LLS the activity of a node is high only in a section of the input space, that is, *localized activation*. This is often accomplished by a Gaussian function, but also box-like functions are used by some models. In some of the models (e.g., self-organizing maps) this also corresponds to locality among the nodes, but the LLS model does not demand this.
- *Learning rules*: Only the nodes with high activity will update their parameters, thus the localized activity will result in *localized learning* [45, 46]. This fact can be used in an *on-line* system [76], that is, where learning must be done incrementally and in real-time, with the results of learning being available soon after each new example is acquired. As the title of the paper indicates, the on-line aspect is important and will be used to select among the variations of the LLS.

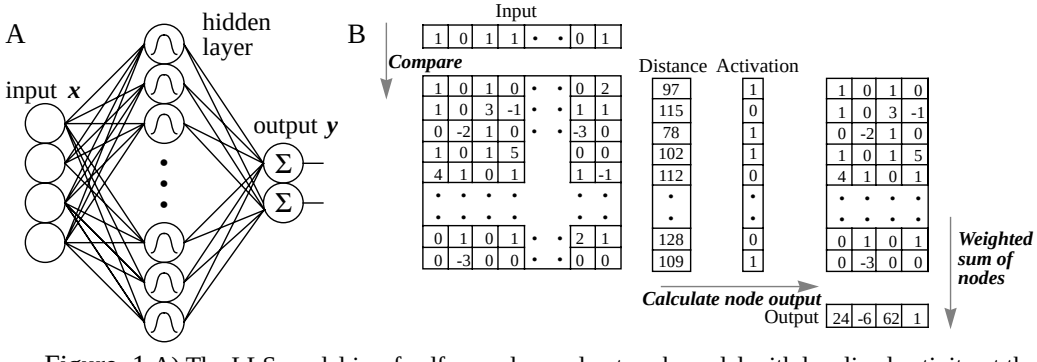

Figure 1 A) The LLS model is a feedforward neural network model with localized activity at the hidden nodes. B) shows the data flow and data organization of the LLS model (feedforward phase).

The LLS algorithm can be divided into two phases, a *feedforward phase* and a *learning phase*. Furthermore the feedforward phase can be divided into three main steps, found in all LLS variations.

1. Calculate the distance r_i between kernel centers (templates) $\boldsymbol{c}_i$ and input $\boldsymbol{x}$, using a distance measure like Euclidean distance, inner product, or Hamming distance.
2. Calculate the node outputs $\varphi(r_i, S_i)$ for nodes i in the hidden layer, where S_i is used to control the size and form of the receptive field. $\varphi(r_i, S_i)$ is known as a *kernel* or *radial-basis function.* A node with a significant output is called a *selected* or *active node*, that is, a node is in the set of active nodes A if $\varphi(r_i, S_i) \geq \alpha$, with $\alpha \geq 0$ being a constant. In many variations of the LLS model $\varphi(r_i, S_i)$ is a Gaussian function with its center located at $\boldsymbol{c}_i$, and a covariance matrix of S_i^{-1}.
3. Calculate the output y as a weighted sum of the (active) nodes, that is

$$y = F(\boldsymbol{x}, \Theta) = \sum_{i \in A} w_i \varphi(r_i, S_i), \quad (1)$$

where $\Theta = \{w_i, \boldsymbol{c}_i, S_i\}_{i=1}^{M}$ are the free parameters. This equation can easily be expanded to multiple outputs as well.

These three steps correspond to the first weight layer, the hidden node layer, and the second weight layer respectively in Figure 1A. In Figure 1B we also illustrate the dataflow and data organization of the LLS. This figure will be the starting point for the discussion in the companion paper on how to map the LLS onto parallel computers.The learning phase adapts the free parameters Θ. The methods for learning are much more differentiated and will only be described in later sections.

3. LLS framework

In this section we introduce the function approximation aspect of LLS. We also discuss methods to formally derive the LLS concept using regularization techniques. The basic concept is then expanded and in the end we show the whole class of LLS variations.

As most signal processing tasks can be seen as functions f mapping an input $x \in \mathbf{S}_{in}$ to an output $y \in \mathbf{S}_{out}$, the function approximation aspect of LLS is of great importance. This will be only briefly described in this section, but a more thorough analysis can be found in [24, 59, 61, 62].

Given a set $D = \{d_p = (\mathbf{x}_p, \mathbf{y}_p) \in \mathbf{S}^d \times \mathbf{S}^m\}_{p=1}^N$ of data, which is a partial specification of a function f belonging to some space of functions defined on $\mathbf{S}^d$, the function approximation problem is to recover the function f, or its approximation, even in the presence of noise.

If both input and output spaces $\mathbf{S}_{in}$ and $\mathbf{S}_{out}$ are real valued, i.e., $\mathbf{S}_{in} = \mathbf{R}^d$ and $\mathbf{S}_{out} = \mathbf{R}^m$, we have the general *function approximation problem* (we sometimes simplify the reasoning to $m = 1$, without loss of generality). If, instead, the output space is binary, i.e $\mathbf{S}_{out} = \mathbf{B}^m$, we have a *classification problem*. Moreover, if both input and output spaces $\mathbf{S}_{in}$ and $\mathbf{S}_{out}$ are binary valued, i.e., $\mathbf{S}_{in} = \mathbf{B}^d$ and $\mathbf{S}_{out} = \mathbf{B}^m$, we have a *Boolean function approximation problem*.

It should be noted that the function approximation problem is ill-posed [10, 80] as there is not enough information in the data available to find a unique solution for the function f. That is, in areas where data are not available nothing can be said about f. By imposing some *a priori* knowledge, which will make the problem well-posed, a solution can be found. Mao and Jain present in [43] a taxonomy of regularization techniques, or ways to describe the *a priori* knowledge, in the ANN field: Type I *Architecture-inherent* which specifies network sizes and topology; Type II *Algorithm-inherent* which specifies learning algorithm, e.g., stopping criteria, noise addition; Type III *Explicitly-specified* which explicitly specifies the stabilizer based on knowledge of the underlying solution, e.g., smoothness or variance constraints. By smooth we mean that small changes in some input parameters determine a correspondingly small change in the output.

LLSs can be derived using all three techniques, but most common are the *architecture-inherent* and the *explicitly-specified*. For the former the structure is specified *a priori*, for instance, deciding on a structure like the one described in Eq. (1). With the explicitly-specified, often referred to as regularization theory, we can choose between fitness to data, e.g., $\sum_{p=1}^N (F(\mathbf{x}_p, \Theta) - y_p)^2$ and the degree of regularization (generalization) $\|\mathrm{P}F(\Theta)\|^2$. The trade-off is controlled by a regularization parameter λ. In the operator P the prior information is incorporated, and therefore P will be problem-dependent, often P is taken to be a differential operator. P is also referred to as a stabilizer in the sense that it stabilizes the solution F. So instead of only using data fitness error we want to minimize:

$$\sum_{p=1}^N (F(\mathbf{x}_p, \Theta) - y_p)^2 + \lambda \|\mathrm{P}F(\Theta)\|^2 \quad \textbf{(2)}$$

The radial basis functions (RBFs) is a powerful group of function approximators which originally has been derived from function approximation theory [8, 61]. The RBF network can

also be derived from regularization theory, as shown by Poggio et al. [58, 59]. By restricting P to be invariant under both rotations and translations the RBF networks can be derived, and by further specifying P, the resulting radial function can be made to be a Gaussian function. Given these restrictions and minimizing Eq. (2) with respect to Θ leads to a solution, an RBF network, where the approximating function $F(\boldsymbol{x}, \Theta)$ is constructed from a set of M radial functions:

$$F(\boldsymbol{x}, \Theta_i) = \sum_{i=1}^{M} w_i \varphi(r_i) \quad \textbf{(3)}$$

As in Eq. (1), $\varphi(r_i)\big|_{i=1}^{M}$ is a set of kernel or radial-basis functions. The distance $r_i = \|\boldsymbol{x} - \boldsymbol{c}_i\|$ where $\|\cdot\|$ denotes a norm, is usually taken to be the Euclidean. The centers (or templates) of the kernels are denoted $\boldsymbol{c}_i$. The free parameters to be set, usually by training, are $\Theta_i = \{\boldsymbol{w}_i, \boldsymbol{c}_i\}$. Typically, the Gaussian $\varphi(r) = e^{-r^2}$ is used as the radial-function. According to regularization theory other radial functions, like multiquadratics, $(r^2 + a^2)^{-1/2}$, or thin-plate splines, $r^2 \log r$, are possible, but as most of them are not conforming to the locality concept they will not be considered further here. This non-conformance also means that some RBF networks are not LLSs. We also find that in regularization theory [59] it is not unusual to add a polynomial to Eq. (3) in the form of $\sum_{i=1}^{n} v_i p_i(\boldsymbol{x})$ $(n \leq d)$ where $p_i\big|_{i=1}^{d}$ is a basis of the linear space $\pi_{k-1}(\boldsymbol{R}^d)$ of algebraic polynomials of degree at most $k-1$ from $\boldsymbol{R}^d$ to $\boldsymbol{R}$, with k given. This however destroys the locality we want. It is still possible to add a bias to Eq. (3) if needed An easy way to accomplish this is by letting one of the kernels always be equal to one.

Other variations of the LLS model are derived in a more architecture-inherent way. A large family of LLS variations is found by varying such things as input and output spaces (Real, Integer or Boolean); distance measures (e.g., cityblock, L_1, or Euclidean, L_2); receptive field (e.g., sphere or elliptic); kernel function (exponential, threshold, min/max etc.); and how to initiate and adapt the free parameters. Many of these variations are well known ANN models by themselves, but by using the LLS concept the connection between these models becomes clearer. In the following two Sections the LLS variations shown in Table 1 will be discussed.

Input	Distance	Receptive Field	φ	$\boldsymbol{c}_i$ initiation	$\Delta\boldsymbol{c}$	Δw	ΔS	Out
R	L_1 (cityblock)	1	exp	Random	Fixed	*Pseudo-inverse*	Fixed	R
I	L_2 (Euclidean)	sI	Radial	Uniform	Gradient	Gradient	Gradient	I
B	L_∞	$s_i I$	Threshold logic unit	Subset of data	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	$\mathrm{diag}\,[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	$\mathrm{diag}\,[s_j]_i$			Incremental addition			
		S_{ij}			Genetic algorithm			
		Hierarchical						
		Sample/Hash						

Table 1 Variations of LLS models. The pseudoinverse is neither local nor on-line, but is commonly used and therefore will be further discussed in Section 4.1

From a theoretical point of view it is interesting to note that when the desired network output is binary and a square error cost function (or a cross entropy cost function) is minimized, the actual network outputs are estimates of the optimal Bayesian conditional probabilities. This is nicely derived, and demonstrated by simulations, for MLP and RBF networks by Richard and Lippmann in [66]. Then the suggested normalized version of Eq. (3) [47]:

$$F(\boldsymbol{x}, \Theta_i) = \frac{\sum_{i=1}^{M} w_i \varphi(r_i)}{\sum_{i=1}^{M} \varphi(r_i)}, \tag{4}$$

becomes natural, if the node outputs are to be interpreted as true probability densities [82]. It has no effect on the classification the network performs, but it does affect the evaluation of the "confidence" in classification. The normalization is also sometimes found to speed up convergence, see for instance Saha et al. [71]. Unfortunately it can also make the network emphasize "outliers" which is clearly undesirable. More importantly, this normalization is global in its nature and therefore does not fit into the LLS concept.

3.1 Number of Kernels

Maybe the most fundamental parameter is the number of kernels M. The original formulation of RBF networks uses the same number of kernels as there is data, that is, $M = N$. This means that if a large training set is used, the number of nodes becomes large. Another problem is that this method is batch oriented, and therefore not suitable for an on-line LLS. For generalizations of RBFs (GRBFs) the number of kernels is reduced to a subset and the number is fixed. Usually this number is set prior to learning in a more or less *ad hoc* manner. The optimal number of kernels depends on the problem at hand, but unfortunately the problem properties are usually not known prior to training.

One possible solution is to use a method which incrementally adds new nodes to the network when needed (to meet some performance criteria). These methods are *structurally adapting* in contrast to the commonly used adaptation of weights. The two basic problems involved are: when to add a new node, and where it should be placed (possibly adjusting other nodes simultaneously). The easiest incremental method is first checking to see if the input is close enough to any of the previously added nodes, and if not, to add a node at the input data position. This is similar to the resource-allocation network (RAN) suggested by Platt [57]. RAN has later been enhanced by Kadirkamanathan and Niranjan [33]. More complicated structural adaptation methods exist (e.g., with deletion) both with topology constraints (Tan [79]) and without (Fritzke [16]). For the Boolean approximation problems a number of interesting structurally adapting methods can be found. One area where it is commonly used is the identification and classification of bacteria, see for instance the idea of cumulative classification in [20, 21].

When the input dimension increases there is a problem with the “curse of the dimensionality”, i.e., the exponentially increasing input space, needing an exponentially increasing number of nodes to have the same coverage of the input space. A number of solutions have been suggested [19, 22, 59, 71, 83] to overcome the dimensionality problem. For instance; transforming the input space, both by globally transforming the input $\boldsymbol{x}$, and by locally transforming the receptive fields of the nodes (see Section 4.2); distributing the centers according to the data distribution (see Section 5.2); or pre-processing the features by any of the dimension reduction methods available, e.g., principal component analysis.

4. The feedforward calculations

For the feedforward phase we find three things that decide locality: distance measurement used, receptive field size and form, and kernel function used. In the next three subsections we will discuss them in detail, using the variations found in Table 1. In subsection 4.4 the calculation of the weighted sum required for the network output will be discussed. In Section 6 a simplified complexity analysis is carried out on both the feedforward phase and the adaptation phase discussed in Section 5.

Throughout this section we will use d as the number of inputs, M as the number of nodes, A as the set of active nodes, and m as the number of outputs.

4.1 Distance measurement

The distance or the similarity between two vectors can be measured in many ways, see for instance [38] where Kohonen discusses the matter. There it is also noted that distance and similarity actually are reciprocal concepts, so in principle we could call distance dissimilarity. Simard et al. have suggested [72, 73] a distance measurement suitable for the LLS model called tangent distance, which can be made locally invariant to any set of the input, and can be computed efficiently. However, in most approximation problems the L_p-norms with $p = 1, 2$ and ∞ are used as a distance measurement. For finite p the L_p-norm in $\boldsymbol{R}^d$ has the value

$$r_i = \|\boldsymbol{x} - \boldsymbol{c}_i\|_p = [\sum_{j=1}^{d} |x_j - c_{ij}|^p]^{1/p}. \quad (5)$$

For $p = 1$ and 2 the L_p-norms can be written as:

$$L_1 \text{ (City block): } r_i = |\boldsymbol{x} - \boldsymbol{c}_i| = \|\boldsymbol{x} - \boldsymbol{c}_i\|_1 = \sum_{j=1}^{d} |x_j - c_{ij}|, \quad (6)$$

$$L_2 \text{ (Euclidean): } r_i = \|\boldsymbol{x} - \boldsymbol{c}_i\| = \|\boldsymbol{x} - \boldsymbol{c}_i\|_2 = (\sum_{j=1}^{d} [x_j - c_{ij}]^2)^{1/2}. \quad (7)$$

The L_∞-norm has the value

$$r_i = \|\boldsymbol{x} - \boldsymbol{c}_i\|_\infty = \max_{1 \le i \le d} |x_j - c_{ij}|. \quad (8)$$

With $\boldsymbol{x} \in \boldsymbol{B}^a$ and $\boldsymbol{c}_i \in \boldsymbol{B}^d$ the *Hamming distance* can be used as a distance measurement, indicating how many bits (positions) of the two vectors that are different. Usually it is calculated as a sum of bits after doing an exclusive-or between $\boldsymbol{x}$ and $\boldsymbol{c}_i$. It is actually equal to both L_1 and $(L_2)^2$ distances, used with binary inputs.

For self-organizing maps (SOM) the *dot-product* $r_i = \boldsymbol{x}^T \boldsymbol{c}_i = \sum_{j=1}^{d} x_j c_{ij}$ often replaces the L_2 distance, typically after normalization of $\boldsymbol{x}$ and $\boldsymbol{c}_i$ to unit lengths, which in fact makes the dot-product proportional to the (negative) L_2 distance as $\|\boldsymbol{x} - \boldsymbol{c}_i\| = \|\boldsymbol{x}\| + \|\boldsymbol{c}_i\| - 2(\boldsymbol{x}^T \boldsymbol{c}_i)$. The search for a node with minimum L_2 distance is correspondingly changed to a search for the maximum dot-product. The dot-product is also regarded as more "neural" than other distance measurements. The Hamming distance is the natural distance for binary input, so it is typically used in SDM and CMAC. The L_2 (Euclid-

ean) distance is the commonly used distance for real-valued inputs. The L_∞ distance has been used by Prager et al. [63, 64] to extend the SDM model to integer input values. The L_1 distance has for instance been used for some hardware implementations of restricted Coulomb energy (RCE) [48].

4.2 Receptive field

The receptive field, that is the part of the input domain which causes a significant output, is determined by a parameter S_i. A general form of distance measure r_i can be defined as $r_i^2 = (\boldsymbol{x} - \boldsymbol{c_i})^T S_i (\boldsymbol{x} - \boldsymbol{c}_i) = \|\boldsymbol{x} - \boldsymbol{c}_i\|_S$ where S_i is a $d \times d$ positive definite matrix. This measure is also named Mahalanobis distance. Note that it implicitly uses the square of the L_2 norm, i.e., $\|\boldsymbol{x} - \boldsymbol{c}_i\|_S = \|\boldsymbol{x} - \boldsymbol{c}_i\|_{2,S}^2$. We can identify S as the inverse of the covariance matrix, given that we use a Gaussian kernel function. The special cases $S_i = diag[s_1, \ldots, s_d]_i$, $S_i = s_i I$ or $S_i = I$ are usually the receptive fields used for LLSs. The parameter S_i determines the receptive field or the support of the function $F(x) = \varphi([\boldsymbol{x} - \boldsymbol{c}_i]^T S_i [\boldsymbol{x} - \boldsymbol{c}_i]) - \alpha$ with $\alpha \geq 0$ being a constant. In other words, the receptive field is a subset of the input domain such that $\varphi([\boldsymbol{x} - \boldsymbol{c}_i]^T S_i [\boldsymbol{x} - \boldsymbol{c}_i])$ takes a value larger than a previously defined constant α.

In the original version of RBF networks the receptive field parameter $s = 1/\sigma^2$ (σ is the standard deviation of the gaussian) was fixed, i.e., $S_i = sI$. Many times it has been set by some heuristics, one such heuristic "global first nearest neighbor" is described by Moody and Darken [47]. The value σ^2 is a global average of the Euclidean distance between all nodes and their closest neighbor $s^{-1} = \sigma^2 = (1/M) \sum_{i=1}^{M} (\min_j \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_S)$. The parameter σ decides the degree of smoothing that will occur. By letting the radius σ approach zero ($s \to \infty$) the operation becomes a table look-up. That is, each input becomes associated with only one node and that node's output.

The form of the receptive field will depend on the kind of basis function used. In the case of a Gaussian function and $S_i = s_i I$, the shape is a hypersphere with a radius determined by s_i. This should be contrasted to the hyperplanes that an MLP generates. When $S_i = diag[s_1, \ldots, s_d]_i$ the shape will be an ellipsoid, the axes of which will coincide with the coordinate axes of the input domain, thus a local rescaling of the input. This type of network is called elliptic basis function (EBF) network and has been studied by Park and Sandberg [55]. If only a global rescaling of the parameters is done $S = diag[s_1, \ldots, s_d]$ these

parameters will reflect the importance of each input dimension [71]. In Figure 2 we see where different versions of S_i fit into the feedforward dataflow.

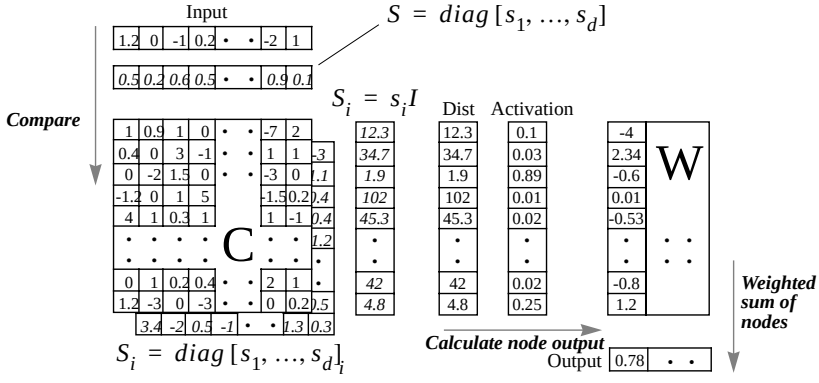

Figure 2 The data flow for the feedforward phase. While calculating the distances (Dist), the inputs are compared to each row in C, using S_i as a weighting or receptive field. Different versions of S_i are shown to indicate where in the dataflow they are used. After calculating the node outputs (Activation) using the distances, the network outputs are calculated as a weighted sum of the node outputs.

Complex receptive fields

Most of the time the forms of receptive fields described above will be sufficient. Still, more powerful forms are available. One example is if non-zero off-diagonal elements are allowed, were $S_i = R_i^T D_i R_i$ with D_i being a diagonal matrix determining the shape of the ellipsoid, and R_i being a rotational matrix determining its orientation. As the number of parameters to be adapted increases dramatically if a general S_i is used, the more specialized forms are often preferred. Therefore, while discussing the number of operations needed, the diagonal form will be used as a reasonable "maximum" number of operations needed.

Another interesting form of receptive fields is found in the conic section function network developed by Dorffner [14]. His model makes it possible to go continuously (via a single parameter) from a linear separation (hyperplane) to a circle (hypersphere) with intermediate types like ellipses and hyperboles. A similar thing would also be possible with a complete second order polynomial (cf. Section 3). However, this would result in a much larger number of free parameters. Unfortunately, it seems that this method needs global information during learning and therefore does not have localized learning.

As problems can contain both high frequency regions (requiring many nodes with small receptive fields) and low frequency regions (requiring fewer nodes but with large receptive fields) methods to cope with this *multiscale* problem might be needed. Poggio and Girosi [59] have suggested a solution based on regularization theory called HyperBF. This model uses a hierarchy of *fixed* receptive fields s_r with different radii. It also makes an interesting connection to Gabor filters and Wavelet neural networks, which some researchers have started to explore, e.g., [7].

In competitive learning, self-organizing maps (SOM), and learning vector quantization (LVQ) [39], the receptive field is reduced to a point, that is, only one node is active (the clos-

est one). Anyhow, a number of researchers [40, 42] have suggested hierarchical variants of these models, both to speed up computations (on sequential computers) and to improve the clustering.

If the input is of high dimension, and a uniform distribution of nodes is used, the number of nodes can be very high. This often shows up in binary models where each real valued input is coded using a multibit encoding. One solution commonly used for binary models like CMAC is to map the input vector to lower dimension using a hashing function, which in fact maps multiple input regions to the same node. Another variation on the same theme is the selected coordinate method suggested by Jaeckel [30] for the SDM model.

4.3 Kernel function

As noted in Section 3, a large number of radial functions can be found using regularization theory [59, 61]. Still the Gaussian function $\varphi(r) = e^{-r^2}$ is the one commonly used. Both its property of having a "nice" derivative, and the many results from probability theory, make the Gaussian function a natural choice. Many of the other radial functions are global in their nature, and can for our purpose be rejected. It can be argued that the Gaussian function really is not local as its tail never goes down to zero. However, for most implementation the activity can be set to zero for nodes far from the input. This is also the result of many commonly used approximations of the exponential function [51].

For the more binary oriented methods a threshold logic unit (TLU) is often used as the kernel function. This gives the receptive field the form of a hypercube instead of a hypersphere typically found in GRBF networks.

In the SOM model by Kohonen [38] the formation of localized responses is attained by using nodes with lateral feedback, that is, a topology among the nodes is assumed. This lateral interaction among the nodes is often described as a "Mexican-hat function". To speed up the computations needed Kohonen suggests a computationally simpler "shortcut" method to achieve the same clustering effect. In the shortcut method the most active node (the winner) is found through a global search and only this node and its topological neighbors will be active, thus, a form of "winner-take-all".

4.4 Network output

The network output y is computed as a weighted sum of the hidden node outputs $y = \sum_{i \in A} w_i \varphi(r_i)$. This is a global reduction operation where all active nodes A in the hidden layer contribute to the output. The degree of locality spans from the case when a winner-take-all operation is used and A only contains one node (the winner), to the case where a non-local kernel function φ is used and A contains all the hidden nodes. As the amount of update is proportional to how close a given node is to the target the latter can be seen as a "softer" form of competition compared to the "hard" competition found in winner-take-all models. In [53] Nowlan suggests that the "soft" competition gives better performance on classification tasks. On a sequential computer the number of operations needed will depend on the number of active nodes contributing to the weighted sum. Thus, the simulation can

run faster if fewer nodes are active. This is however not necessarily true for parallel computer implementations.

For parallel computer implementations reduction operations are of general concern as they require extra communication means. Anyhow, for many LLS variations the reduction operations seem unavoidable, and in the companion paper [51] we discuss way to support these operations on parallel computers. Note for instance that to find the winner for a winner-take-all computation a global minimum is required, that is, a reduction using the minimum operation is required.

5. Setting the Free Parameters

The free parameters to be set by training are $\Theta_i = \{w_i, \boldsymbol{c}_i, S_i\}\big|_{i=1}^{M}$, where M is the number of kernels. In the following subsections, one for each of the free parameters, we describe the adaptation methods typically used.

Many of the methods can be used together and/or used in sequence. A typical scenario in an off-line situation is to start with a subset of data as initiation of centers, and use a clustering method to start adapting the centers, e.g., competitive learning. After this initial clustering a "global first nearest neighbor" heuristic is used to set the receptive field sizes and the weights are set using the pseudoinverse described below. Finally all the free parameters are adapted using a gradient method. There are of course many other ways to combine the different methods of adaptation described below.

We also note that many of the methods used to speed up learning for MLPs can be applied for LLS variations that use gradient descent learning (especially GRBFs). Such methods are for instance addition of a momentum term [69], conjugate gradient learning, or other higher-order optimization methods [24].

5.1 Adaptation of $\boldsymbol{w}_i$

To simplify the derivation of an update rule for w_i we temporary assume that all M nodes are included in the active set A. Then for each training example p, we can write $y_p = F(\boldsymbol{x}_p, \Theta_i) = \sum_{i=1}^{M} w_i \varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|) = \Phi_p^T \boldsymbol{w}$, where $\Phi_p = [\varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_1\|), \ldots, \varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_M\|)]^T$ and $\boldsymbol{w} = [w_1, \ldots, w_M]^T$. The problem is to find $\boldsymbol{w}$ given the N simultaneous equations, that is, solving:

$$Y = \Phi \boldsymbol{w}, \tag{9}$$

where $\Phi_{pi} = \varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|)$. The classical solution to Eq. (9) is to minimize the sum of the squared error, that is

$$\mathbf{E} = \|\Phi \boldsymbol{w} - Y\|^2, \tag{10}$$

which is the same as Eq. (2) with the regularization parameter $\lambda = 0$. The gradient

$$\frac{\partial \mathbf{E}}{\partial \boldsymbol{w}} = 2\Phi^T (\Phi \boldsymbol{w} - Y), \tag{11}$$

can either be used "as is" in a gradient search method, or a closed form can be derived by setting the partial derivative to zero. This yields the condition $\Phi^T \Phi \boldsymbol{w} = \Phi^T \boldsymbol{y}$, which has the pseudo inverse solution:

$$\boldsymbol{w} = (\Phi^T \Phi)^{-1} \Phi^T Y = \Phi^{\dagger} Y, \tag{12}$$

where $\Phi^{\dagger}$ is the Moore-Penrose pseudoinverse of Φ. If no duplicate kernels exist (i.e., $\boldsymbol{x}_p \neq \boldsymbol{x}_q$ if $p \neq q$) then $(\Phi^T \Phi)$ is invertible. This inverse can be computed in $O(M^3)$ operations using a singular value decomposition method[1]. Because of the many operations (hard

to calculate in an on-line fashion) and its non-local nature, we are interested in other methods, despite the fact that Eq. (12) is optimal in the least-square sense.

The direct gradient search method will use Eq. (11) to update $\boldsymbol{w}$ as

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \eta_w \frac{\partial \mathbf{E}}{\partial \boldsymbol{w}} \tag{13}$$

where η_w is the learning rate.

By updating $\boldsymbol{w}$ after each training pattern presented, there is a deviation from the true gradient descent in $\mathbf{E}$. This method, sometimes called incremental or on-line gradient learning, is related to stochastic gradient methods [23]. For the learning to converge, special care is needed, for instance, the learning rate should be reduced compared with the batch method, and the order of the training patterns becomes important. Still this incremental gradient method has been successfully used both for MLPs [69] and LLSs [23], and it is an important concept for any on-line gradient descent method.

Using a weighted Euclidean distance measurement the per-pattern error can be written as

$$\mathbf{E}_p = e_p^2 = (F(\boldsymbol{x}_p, \Theta_i) - y_p)^2 = \left(\sum_{i=1}^{M} w_i \varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S) - y_p\right)^2, \tag{14}$$

and (for scalar output) the update equation for the weights w_i can be written as

$$w_i(t+1) = w_i(t) - 2\eta_w e_p \varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S). \tag{15}$$

Under the assumption that all M nodes are included in the active set the total number of operations for updating all the w_i weights is $2M$ multiplications and M additions/subtractions. If instead a subset A of nodes is active Eq. (15) is only applied to those weights where $i \in A$, and the total number of operations for updating w_i is reduced accordingly. Note that the node output $\varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S)$ is already calculated in the feedforward process, and that the factor 2 can be incorporated into the learning rate η_w.

Hebbian rule or occurrence

It is also possible to use the Hebbian learning rule described by Hebb 1949 in [25]. There he postulates that a connection should be strengthened by any co-occurrence between input (presynaptic) and output (postsynaptic) activities. As a special case of this rule, suitable for the "output nodes" of an LLS, we can write

$$w_i(t+1) = w_i(t) + \eta_w y_p \varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S). \tag{16}$$

In the case of multiple output nodes, the weight update becomes an addition of an outer-product matrix (the outer-product is between the vector consisting of the kernel outputs and

1. As Aho et al. [1] have shown that taking the matrix inverse is no harder than matrix multiplication, and as Coppersmith and Winograd [9] have developed a matrix multiplication algorithm with an asymptotic running time of $O(M^{2.376})$, there must be a matrix inversion algorithm with an asymptotic running time less that $O(M^3)$. However, for practical purposes, especially on parallel hardware, we need to use an algorithm that on a sequential machine has an asymptotic running time of $O(M^3)$.

the output vector). In the case of binary node activity (TLU kernel function) and binary outputs, this update equation results in a counting of co-occurrences between node activity and output. This update rule is used in SDM models with the modification that the output is taken to be −1 and 1 instead of 0 and 1.

The Hebbian rule in its original form will lead to an unlimited growth of the weights, which is clearly undesirable. To avoid this problem some sort of limit on the weights, or their growth, is needed, see for instance [24] for a discussion on various methods. For the SDM model the weights are assumed to have a soft overflow, where any attempt to increment a weight above the maximum value will keep the weight at the maximum.

5.2 Position of the kernel centers c_i

Both adaptive and non-adaptive methods to set suitable centers exist. If no *a priori* information exists, two natural methods are to distribute the centers either randomly or uniformly over the input space. As soon as some training data is available this new information can be used to decide better center positions. The simplest method is to use the data (or a subset of the data) itself as center positions. In Table 2 some of the variations to initiate the position centers are listed.

Position method	ANN model
Random	SDM [34], SOM and LVQ [39]
Uniform	CMAC [4]
Data or a subset	RBF [61], SDM [31], probabilistic neural networks (PNN) [75]

Table 2 Methods to initiate position centers used in different ANN models.

5.3 Adaptation of kernel centers

The adaptation of centers can be divided into unsupervised or supervised methods. The unsupervised methods usually are in the form of *competitive learning* (CL), like k-means clustering or SOM. All of these unsupervised methods will distribute the nodes according to the input density (with varying success), and for many problems this will be sufficient or possibly close to optimal. The SOM suggested by Kohonen [37, 39] uses a spatial topology constraint to aid the clustering effect of the method. Among the many variations of CL the rival penalized competitive learning (RPCL) will later serve as an example.

If the network is to be used for classification, it is not certain that the best node distribution is the input distribution. Both clustering and principal component analysis (PCA) can fail for cases where there is an uneven sampling of classes.

However, if training examples with desired outputs are available a supervised method can be used. For example, methods like LVQ [39] or *gradient descent*. In Section 5.3.1 the gradient descent method will be discussed. In [18] Ghosh and Chakravarthy show that under

some general conditions the positions of the centers obtained by SOM are similar to those for a GRBF network used in an unsupervised fashion (constant output), and updated with a per-pattern gradient descent method. The clustering effect of centers using gradient descent GRBF network have also been noted by Poggio and Girosi [58]. As discussed in Section 3.1 a number of *incremental methods* exists. For the SDM model Rogers has suggested [67] a g*enetic algorithm* [27] as a way to adapt the kernel centers. All the methods mentioned above have been used by different LLS variations. References to some of them are shown in Table 3.

Position method	ANN model
Competitive Learning	RBF [47], SDM [70], RPCL + RBF [84], SOM and LVQ [39], Counter Propagation [26]
Genetic algorithms	SDM [67]
Incremental addition	RCE [6], Growing-RBF [16], RAN [57]
Gradient methods	GRBF [58]

Table 3 Methods to adapt position centers used in different ANN models.

5.3.1 Gradient descent

As an example of adaptation of the kernel centers we here describe the gradient descent method. For this method the update equation for the kernel centers can be found using the same approach as for w_i in Eq. (15). The per-pattern kernel center $\boldsymbol{c}_i$ gradient update equation can be written as

$$\boldsymbol{c}_i(t+1) = \boldsymbol{c}_i(t) + 4\eta_c w_i e_p \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S) S_i (\boldsymbol{x}_p - \boldsymbol{c}_i) \quad \textbf{(17)}$$

where φ' is the derivative of φ. We note that $(\boldsymbol{x}_p - \boldsymbol{c}_i)$ and $\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S$ are already calculated in the feedforward phase and can be reused (if storage is available). If the kernel function is a Gaussian function it is not even needed to calculate $\varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S)$, as the derivative of an exponential function is equal to itself. If, for instance, a diagonal S_i is used the total number of operations for updating all the $\boldsymbol{c}_i$ vectors are $(3 + 2d)M$ multiplications and dM additions. This is under the assumption that all M nodes are included in the active set A. If instead a subset of nodes is active the Eq. (17) is only applied to those weights where $i \in A$, and the total number of operations for updating $\boldsymbol{c}_i$ is reduced accordingly.

When using gradient descent, a careful initialization of the nodes is needed. This is especially true when localized learning is used as the network might end up with many nodes too far from any training data to become part of the learning process. One solution to this problem is to initiate the network with a selection of the data, another is to use a clustering method (like RPCL) to find good starting points.

5.3.2 Rival penalized competitive learning (RPCL)

As an example of a CL method we have chosen the rival penalized competitive learning (RPCL) developed by Xu et al. [84]. This expands the basic CL algorithms in two areas, it introduces a rivalry among the winner node and the runner up, and it uses a frequency compensation (conscience) to get all nodes actively taking part in the competition.

The update equations for RPCL are:

$$\boldsymbol{c}_i(t+1) = \boldsymbol{c}_i(t) + u_i(\boldsymbol{x}_p - \boldsymbol{c}_i) \tag{18}$$

$$\text{where } u_i = \begin{cases} \eta_k & \text{if } i = k \text{ such that } \gamma_k \|\boldsymbol{x}_p - \boldsymbol{c}_k\|_2 = \min_j \gamma_j \|\boldsymbol{x}_p - \boldsymbol{c}_j\|_2, \\ -\eta_r & \text{if } i = r \text{ such that } \gamma_r \|\boldsymbol{x}_p - \boldsymbol{c}_r\|_2 = \min_{j \neq k} \gamma_j \|\boldsymbol{x}_p - \boldsymbol{c}_j\|_2, \\ 0 & \text{otherwise,} \end{cases} \tag{19}$$

for $i = 1, \ldots, M$ and $0 \leq \eta_k, \eta_r \leq 1$. Usually $\eta_k(t) \gg \eta_r(t)$. Moreover, $\gamma_j = n_j / (\sum_{i=1}^{M} n_i)$ (which is equal to n_j / N), and n_i is the cumulative number of occurrences of $u_i = 1$. We see that the parameter γ_j functions as a receptive field size parameter. As shown by Xu et al. [84], this simple but powerful update rule performs very well in an LLS.

Note the similarity between Eq. (17) and (18) where u_i corresponds to $4\eta_c w_i e_p \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S) S_i$. We can see $w_i e_p$ as the credit assignment factor missing in CL methods and $\varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S) S_i$ as a locality assignment similar to u_i in Eq. (19).

The total number of operations for updating all the $\boldsymbol{c}_i$ vectors, which include a feedforward sequence, is $M(d+1) + 2d$ multiplications and $(2d+1)M + 2d - 2$ additions (if partial results are reused). This leads us to conclude that the gradient and RPCL methods use approximately the same number of operations per pattern.

5.4 Adaptation of S_i

As noted in Section 4.2 the receptive field sizes can be fixed either to a scalar or a set in a hierarchy. Among the adaptive methods the gradient descent method is the most common, but also the more *ad hoc* method of RCE adaptation have been used as an LLS variation. These two variations are discussed below.

Gradient descent

Using the same approach as for w_i in Eq. (15) and $\boldsymbol{c}_i$ in Eq. (17) the gradient update equation for S_i can be written as:

$$S_i(t+1) = S_i(t) - 2\eta_S w_i e_p \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_{S_i}) Q_{pi}, \tag{20}$$

where φ' is the derivative of φ, and $Q_{pi} = (\boldsymbol{x}_p - \boldsymbol{c}_i)(\boldsymbol{x}_p - \boldsymbol{c}_i)^T$ is the outer-product of the distance. If $S_i = diag[s_{i1}, \ldots, s_{id}]$ then Eq. (20) can be simplified (with respect to the number of calculations needed) to

$$S_i(t+1) = S_i(t) - 2\eta_S w_i e_p \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_{S_i}) (diag[(\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})^2, \ldots, (\boldsymbol{x}_{pd} - \boldsymbol{c}_{id})^2]). \tag{21}$$

Likewise if $S_i = s_i$ we can simplify Eq. (20) to

$$S_i(t+1) = S_i(t) - 2\eta_S w_i e_p \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_{S_i}) (\boldsymbol{x}_p - \boldsymbol{c}_i)^T (\boldsymbol{x}_p - \boldsymbol{c}_i). \quad (22)$$

If global rescaling is used we get an update such as

$$S_i(t+1) = S_i(t) - 2\eta_S \sum_{i=1}^{M} \Delta S_i, \quad (23)$$

where for the diagonal form we have

$$\Delta S_i = w_i e_p \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_{S_i}) (diag [(\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})^2, \ldots, (\boldsymbol{x}_{pd} - \boldsymbol{c}_{id})^2]), \quad (24)$$

and similarly for the scalar form

$$\Delta S_i = w_i e_p \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_{S_i}) (\boldsymbol{x}_p - \boldsymbol{c}_i)^T (\boldsymbol{x}_p - \boldsymbol{c}_i). \quad (25)$$

RCE (Restricted Coulomb energy) adaptation

Another form of receptive field size adaptation is found in the RCE method [29, 65]. It is a structurally adapting model similar to the ones discussed in Section 3.1. However, in this method the node positions usually are fixed at input data positions, while the receptive field sizes are adapted. Three update rules can be described, where the simplest rule applies to the case where the input is correctly classified, and nothing needs to be done. If no node is close enough to the input, a new node is added (with a certain radius). And in the case of incorrect classification the radii of the conflicting nodes are reduced until they become inactive, and a new node is added at the data position.

6. Complexity analysis

Due to the many variations possible a thorough analysis of the complexity is not performed in this paper. This is also best done while discussing different mappings to parallel computer hardware. A discussion on suitable mappings and complexity analysis is found in Part II [51]. Instead we give an example using a Euclidean distance, a hyperbolic receptive field ($\mathrm{diag}\,[s_k]_i$), a Gaussian kernel, and gradient learning in Eq. (3), (15), (17), and (21). This is a common configuration and gives, as discussed earlier, a reasonable upper limit on the number of operations needed.

Variation Type	Variation	Result	Operation	Additions	Mult.	Other ops.
Distance	L_2	δ_k	$(x_k - c_{ik})^2$	d	d	
Receptive field	$\mathrm{diag}\,[s_j]_i$	r_i, (r_i^2)	$\sum_k s_k \delta_k$	$d-1$	d	
Kernel function	exp	$\varphi(\ )$	$\exp(-r_i^2)$			2-10 ops.
Output		$\boldsymbol{F}(\boldsymbol{x}, \Theta)$	$\sum_i w_i \varphi(\ )$	$m(M-1)$	mM	
Error		$\boldsymbol{e}_p$	$\boldsymbol{F}(\boldsymbol{x}, \Theta) - \boldsymbol{y}$	m		
$\Delta \boldsymbol{w}$	Gradient	$\Delta \boldsymbol{w}$	$\eta_w \boldsymbol{e}_p \varphi(\ )$		$1+m$	
		$\boldsymbol{w}_i(t+1)$	$\boldsymbol{w}_i(t) + \Delta \boldsymbol{w}$	m		
$\Delta \boldsymbol{c}$	Gradient (L_2, exp, $\mathrm{diag}\,[s_j]_i$)	ε_{ip}	$\sum_{j=1}^{m} e_{pj} w_{ij}$	$m-1$	m	
			$\varphi'(-r_i^2)$			2-10 ops.
		$\partial r^2 / (\partial c)$	$S_i (\boldsymbol{x}_p - \boldsymbol{c}_i)$		d	
		$\Delta \boldsymbol{c}$	$\eta_c \varepsilon_{ip} \varphi'(-r_i^2) \frac{\partial r^2}{\partial c}$		$1+d$	
		$\boldsymbol{c}_i(t+1)$	$\boldsymbol{c}_i(t) + \Delta \boldsymbol{c}$	d		
ΔS	Gradient: (L_2, exp, $\mathrm{diag}\,[s_j]_i$)	$\partial r^2 / (\partial S)$	$(\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})^2$		d	
		ΔS	$\eta_S \varepsilon_{ip} \varphi'(-r_i^2) \frac{\partial r^2}{\partial S}$		$1+d$	
		$S_i(t+1)$	$S_i(t) + \Delta S$	d		

Table 4 The number of operations needed for the feedforward phase of an LLS variation using a Euclidean distance, a hyperbolic receptive field ($\mathrm{diag}\,[s_k]_i$), a Gaussian kernel, and gradient learning. For all operations except for the gray row, the number of operations is found by multiplying by M.

Note that the node output $\varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S)$ is already calculated in the feedforward process, and that the factor 2 or 4 can be incorporated into the learning rates η. While updating S_i we try to reuse as many of the previously calculated partial results as possible, that is, $\varepsilon_{ip}\varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S^2)$, $(\boldsymbol{x}_p - \boldsymbol{c}_i)^T (\boldsymbol{x}_p - \boldsymbol{c}_i)$, and $(x_{pj} - c_{ij})$ are reused.

There are M computations of the kernel function. Depending on which type of approximation is used, and if special support for its calculation exists in the hardware, this will translate to different numbers of primitive operations. To calculate the network output for an LLS using a full "tail" mM multiplications are needed. Having localized activity with $|A|$ nodes active only $|A|m$ multiplications are needed. If the normalization in Eq. (4) is used another $M-1$ additions and M divisions are required.

7. Similarities and Differences Between Different LLS Variations

Within the LLS concept we have already identified a number of common ANN algorithms. In the following subsections we will study two of these ANN models using Table 1 as a starting point. In Table 5 and Table 6 we show the basic form of GRBF and SDM models, together with a number of their variations, all contained within the concept of LLSs. These tables show the close connection among a number of ANN models and variations. We can also identify new variations of old ANN models. Some of the other ANN models recognized as LLS variations are discussed in Appendix A.

The main or basic form is indicated by black ellipses and the variations are marked with gray.

7.1 Radial Basis Function, and generalizations

In Table 5 we show the original formulation of an RBF network [61] in comparison with a number of variations suggested by Poggio and Girosi [59]. These variations concern mainly the initiation and adaptation of the free parameters as discussed in Section 5.

Input	Distance	Receptive Field	φ	$\boldsymbol{c}_i$ initiation	$\Delta\boldsymbol{c}$	Δw	ΔS	Out
R	L_1 (cityblock)	1	exp	Random	Fixed	Pseudo-inverse	Fixed	R
I	L_2 (Euclidean)	sI	Radial	Uniform	Gradient	Gradient	Gradient	I
B	L_∞	$s_i I$	Threshold logic unit	Subset of data	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	$\mathrm{diag}[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	$\mathrm{diag}[s_j]_i$			Incremental addition			
		S_{ij}			Genetic algorithm			
		Hierarchical						
		Sample/Hash						

RBF [61]

Generalizations [58, 59]

Table 5 The original RBF network and the generalizations suggested by Poggio and Girosi [58, 59] The black circles are the default value or main model for each column.

7.2 Sparse Distributed Memory

One of the binary oriented models (for the network input and output) is SDM. The binary nature of the input makes it natural to use a Hamming distance. SDM uses a "radius of activation" ($S = s_i I$) as receptive field, and a threshold as kernel function. The radius is chosen to allow only a small number of nodes to be active at the same time.

The original SDM model [34] used a random, and fixed, distribution of kernel centers. This is far from optimal if the input data is unevenly distributed. As this is common when natural data is used a number of variations have been developed [36]. Many of the variants are shown in Table 6. It is interesting to see that almost all the variations of SDM can be mapped into this table of LLS variations. In [31] Joglekar suggests the usage of data or subsets of data to initiate the kernel centers. As we noted in Section 4.2 Jaeckel's selected coordinate method [30] is an idea similar to the hash coding of the CMAC. Rogers [67] has suggested using genetic algorithms, and Saarinen et al. competitive learning [70], to improve the kernel positions. Pohja and Kaski [60] discuss methods to set the receptive field sizes to force all nodes to participate an equal number of times. In [35] Kanerva heuristically employs a global rescaling to adjust for different importances of the inputs.

Input	Distance	Receptive Field	φ	$\boldsymbol{c}_i$ initiation	$\Delta\boldsymbol{c}$	Δw	ΔS	Out
R	L_1 (cityblock)	1	exp	Random	Fixed	Pseudo-inverse	Fixed	R
I	L_2 (Euclidean) [64]	sI [60]	Radial	Uniform [31]	Gradient [70]	Gradient	Gradient	I
B	L_∞	$s_i I$ [35]	Threshold logic unit	Subset of data [31]	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	diag $[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	diag $[s_j]_i$			Incremental addition [67]			
		S_{ij}			Genetic algorithm			
		Hierarchical [30]						
		Sample/Hash						

■ SDM [34]

■ SDM Generalizations

Table 6 Variations of the SDM model. The black circles are the default value or main model for each column.

Some of the natural extensions to the SDM model found by studying Table 6 are the use of exponential kernel function, the use of topology for the competitive learning variation, the use of local rescaling of input ($S_i = \text{diag}\,[s_j]$), and the use of a RCE-like algorithm to adapt S_i for GRBF networks.

7.3 Comparing LLSs to multilayer perceptrons (MLPs)

The main difference between an LLS and the much used feedforward network MLP, which is not an LLS, is that while the LLS is computing a local approximation, the MLP computes a global approximation. Many times the LLS variations have been shown to learn faster (two orders of magnitude faster is not uncommon) [47], while the global approximation capability makes the MLP a better approximator in regions where none or very few training data exist. On the other hand, the local learning seems to be better to learn disjoint regions and is less sensitive to the order of the training patterns shown to the network during training. When comparing the performance of these networks they seem capable of similar accuracy, at least if both are trained correctly.

Other differences between LLSs and MLPs are that while LLSs have one hidden layer the MLPs use one or many hidden layers; LLSs commonly use Euclidean distance while MLPs use dot-product; and LLSs use different kinds of nodes for hidden layer and output layer whereas MLPs commonly have the same kind of nodes for all layers. The structure of the LLS, especially the local activity and learning concept as well as the expanded representation, seems to be more biologically “plausible” than a MLP with error back propagation (BP).

A drawback with an LLS is that many times there are more nodes needed in the hidden layer to achieve a certain degree of accuracy. This should not come as a surprise, since an expanded representation is used for an LLS. This can however lead to a slower network when (and if) the training is completed and only a feedforward phase is running (i.e., not learning on-line), at least on a sequential computer.

Another drawback that many LLS variations suffer from (maybe even more than the MLP) is the *curse of dimensionality*, referring to the exponential increase in the number of hidden nodes with the dimension of the input space. This can to some degree be reduced (some of the methods are mentioned in Section 3.1) but a universal solution to this problem has not yet been suggested and is therefore an area needing more research.

8. Conclusion

We have defined a “superclass” of artificial neural networks (ANNs), called localized learning systems (LLSs), with the characteristic features of feedforward network, expanded representation, localized activity, and localized learning. The LLS concept incorporates both binary and non-binary input; both simple and complex receptive fields; both smooth kernel functions and threshold units; and a number of variations on how to adapt the free parameters, like competitive learning and gradient methods. It also incorporates methods that use structural adaptation instead of weight adaptation.

One objective of this paper is to study these different variations of the LLS model in detail. Additionally we show how a number of well known ANN models are connected through the concept of LLS. Another important objective for this paper is to lay a foundation for an analysis of parallel computer implementations of LLSs to be carried out in Part II [51]. To better suit applications where continuous learning is needed, the possibility to do on-line computations has been emphasized, thus restricting the possible variations.

Our description of LLS variations also makes it easy to see and suggest new variations to many of the constituent ANN models. One such variation would be the use of local rescaling of input for SDM ($S_i = \text{diag}\,[s_j]_i$), another would be to use a RCE-like algorithm to adapt S_i for GRBF networks. For the other ANN models in the LLS class, many new variations can be found as well.

There are still some investigations that should be done to make the picture of LLSs complete, for instance, the choice of kernel function needs to be further studied to find truly local receptive fields while keeping the approximation performance high. Another important aspect to be investigated is the introduction of multi-modular networks with many cooperating (and competing) LLS modules. One such model is the “hierarchy of experts” model described by Jordan and Jacobs in [32]. In [13] Davis, Nordström and Svensson note a need for a modular style of computer architecture to match the modular structure of these ANN algorithms. We hope to return to these and other subjects regarding LLSs in a forthcoming paper.

The natural and massive parallelism found in LLSs, together with the locality of activity and learning, and the usefulness of the constituent ANN models makes it very interesting to find hardware suitable for LLSs. In part II we will continue our analysis of LLSs but then the goal is to suggest a suitable (parallel) computer architecture for this class of networks.

8.1 Acknowledgments

The author would like to thank Prof. Bertil Svensson Chalmers University of Technology, Assoc. Prof. Anders Lansner Royal Institute of Technology (KTH), Assoc. Prof. Timo Koski Luleå University of Technology, Assoc. Prof. Lennart Gustafsson Luleå University of Technology, and Lic. Tech. Per Ödling Luleå University of Technology for valuable discussions.

Appendix A

In Section 7 we discussed two ANN algorithms in the context of LLS. In this appendix some of the other well known ANN algorithms in the LLS class will be studied, using Table 1 as a starting point. In Table 7 to Table 10 these ANN models are described using our LLS framework.

The main or basic form of a certain ANN model is indicated by black ellipses while reported variations of that ANN model are marked with various forms of gray.

A.1 Pre-RBF, RBF, and generalizations of RBF

The original form of radial basis function (RBF) networks, where every kernel is located at a data position, is closely related to the *ad hoc* probability density function estimation methods of Parzen windows, and potential functions [15, 56]. The difference is mainly how to derive the kernel weighting w_i. Specht [75] has extended the ideas of Parzen windows to a class of neural networks called probabilistic neural networks (PNN). These models and their different variations are shown in Table 7.

In section 7.1 we showed the original formulation of RBFs in comparison with a number of variations suggested by Poggio and Girosi and others. These variations concern mainly the initiation and adaptation of the free parameters as discussed in Section 5. This can be compared to the algorithms found in Table 7 where the variations only concern the distance measurement, the receptive field, and the radial function.

Input	Distance	Receptive Field	φ	$\boldsymbol{c}_i$ initiation	$\Delta\boldsymbol{c}$	Δw	ΔS	Out
R	L_1 (cityblock)	1	exp	Random	Fixed	Pseudo-inverse	Fixed	R
I	L_2 (Euclidean)	sI	Radial	Uniform	Gradient	Gradient	Gradient	I
B	L_∞	s_iI	Threshold logic unit	Subset of data	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	$\text{diag}[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	$\text{diag}[s_j]_i$			Incremental addition			
		S_{ij}			Genetic algorithm	■ Parzen window [56]		
		Hierarchical				■ Potential function [15]		
		Sample/Hash				■ PNN [75]		

Table 7 Parzen, Potential function, probabilistic neural networks (PNN)

A.2 Competitive learning

Competitive learning (CL) is commonly used to update kernel positions and has been discussed in Section 4.3 and 5.3. One of the CL models, self-organizing map (SOM), has been extended with a linear output layer which is trained in a supervised fashion, to become the so called Counter Propagation (CP) model [26]. This model is the base model in Table 8. In this table we have also shown two variations of structurally adapting CL, called growing cell by Fritzke [16].

Restricted Coulomb energy (RCE), developed by Reilly [65], is another form of structural adaptation where new nodes are incrementally added at data positions and the receptive field is adapted to always classify seen data correctly, as described in the end of Section 5.4. In Table 9 we see a description of the original RCE and two variations [48, 74] which are generalizations of the original model. These variations are also done to fit better into a parallel computer concept and will be discussed in more detail in Part II.

Input	Distance	Receptive Field	φ	c_i initiation	Δc	Δw	ΔS	Out
R	L_1 (cityblock)	1	exp	Random	Fixed	Pseudo-inverse	Fixed	R
I	L_2 (Euclidean)	sI	Radial	Uniform	Gradient	Gradient	Gradient	I
B	L_∞	s_iI	Threshold logic unit	Subset of data	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	diag $[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	diag $[s_j]_i$			Incremental addition			
		S_{ij}			Genetic algorithm	■ SOM and CP [26, 39]		
		Hierarchical				■ Growing Cell [16]		
		Sample/Hash				■ Growing Cell RBF [16]		

Table 8 Competitive Learning

Input	Distance	Receptive Field	φ	c_i initiation	Δc	Δw	ΔS	Out
R	L_1 (cityblock)	1	exp	Random	Fixed	Pseudo-inverse	Fixed	R
I	L_2 (Euclidean)	sI	Radial	Uniform	Gradient	Gradient	Gradient	I
B	L_∞	s_iI	Threshold logic unit	Subset of data	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	diag $[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	diag $[s_j]_i$			Incremental addition			
		S_{ij}			Genetic algorithm	■ RCE [65]		
		Hierarchical				■ RCE (Intel/HCE) [48]		
		Sample/Hash				■ RCE (TI) [74]		

Table 9 Restricted Coulomb Energy (RCE)

A.3 Cerebellar Model Arithmetic Computer (CMAC).

The two models that are binary oriented (for the network input and output) are SDM (discussed in section 7.2) and CMAC. The binary nature of the input makes it natural to use a Hamming distance. Both methods use a “radius of activation” as receptive field, and a threshold as radial function. The radius is chosen to let only a small number of nodes be active at the same time.

What makes CMAC unique is the method to initiate the kernel centers to be uniformly distributed, cf. Table 10. As a consequence the large number of hidden nodes (especially for high dimensional input) have forced the method to include a hashing function between the “logic” kernel position and the “physical” position.

Moody [45] extends the CMAC concept by using radial functions with graded response instead of the threshold units usually used by the CMAC method. He also suggest a hierarchy of receptive fields. This makes this model a precursor to the HyperBF model previously mentioned, in the discussion of complex receptive fields, in Section 4.2.

Input	Distance	Receptive Field	φ	$\boldsymbol{c}_i$ initiation	$\Delta\boldsymbol{c}$	Δw	ΔS	Out
R	L_1 (cityblock)	1	exp [45]	Random	Fixed	Pseudo-inverse	Fixed	R
I	L_2 (Euclidean)	sI	Radial	Uniform	Gradient	Gradient	Gradient	I
B	L_∞	s_iI	Threshold logic unit	Subset of data	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	diag $[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	diag $[s_j]_i$			Incremental addition			
		S_{ij} [45]			Genetic algorithm			
		Hierarchical				■ CMAC [4]		
		Sample/Hash				■ Generalizations		

Table 10 Cerebellar Model Arithmetic Computer (CMAC)

References

[1] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] Albus, J. S., "A theory of cerebellar function," *Mathematical Biosciences*, vol. 10, no. 1/2, pp. 25-61, 1971.

[3] Albus, J. S., "A new approach to manipulator control: the cerebellar model articulation controller (CMAC)," *American Society of Mechanical Engineers, Transaction G (Journal of Dynamic Systems, Measurement, and Control)*, vol. 97, no. 3, pp. 220-227, 1975.

[4] Albus, J. S., *Brains, Behavior, and Robotics*, Petersborough, NH, USA: BYTE/McGraw-Hill, 1981.

[5] Bachkirov, O. A., E. M. Braverman and I. B. Muchnik, "Potential function algorithms for pattern recognition learning machines," *Automation and Remote Control*, vol. 25, pp. 629-631, 1964.

[6] Bachmann, C. M., L. Cooper, A. Dembo and O. Zeitouni, "A relaxation method for memory with high storage density," *Proceedings of the National Academy of Sciences*, vol. 84, pp. 7529-7531, 1987.

[7] Bakshi, B. R. and G. Stephanopoulos, "Wavelets as basis functions for localized learning in a multi-resolution hierarchy," in *IJCNN International Joint Conference on Neural Networks*, Baltimore, MD, USA, 1992, vol. 2, pp. 140-145.

[8] Broomhead, D. S. and D. Lowe, "Multivariate functional interpolation and adaptive networks," *Complex Systems*, vol. 2, pp. 321-355, 1988.

[9] Coppersmith, D. and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, 1987, pp. 1-6.

[10] Courant, R. and D. Hilbert, *Methods of mathematical physics*, vol. 2 London, England: Interscience, 1962.

[11] Cover, T. M., "Geometric and statistical properties of systems of linear inequalities with applications in pattern recognition," *IEEE Transaction on Electronic Computers*, vol. EC-14, pp. 326-334, 1965.

[12] Cybenko, G., "Approximation by superpositions of sigmoidal function," *Mathematics of Control, Signals, and Systems*, vol. 2, pp. 303-314, 1989.

[13] Davis, E. W., T. Nordström and B. Svensson, "Issues and applications driving research in non-conforming massively parallel processors," in *Proceedings of the New Frontiers, a Workshop of Future Direction of Massively Parallel Processing*, I. D. Scherson Ed., McLean, Virginia, 1992, pp. 68-78.

[14] Dorffner, G., "A unified framework for MLPs and RBFs: introducing conic section function networks," *Cybernetics and Systems*, vol. 25, no. 4 to appear, 1994.

[15] Duda, R. O. and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, Inc., 1973.

[16] Fritzke, B., "Growing cell structures — a self-organizing network for unsupervised and supervised learning," Tech. Rep. TR-93-026, International Computer Science Institute, 1993.

[17] Funahashi, K.-I., "On the approximate realization of continuous mapping by neural networks," *Neural Networks*, vol. 2, no. 3, pp. 183-192, 1989.

[18] Ghosh, J. and S. V. Chakravarthy, "The rapid kernel classifier: a link between the self-organizing feature map and the radial basis function network," *Journal of Intelligent Material Systems and Structures*, vol. 5, no. 2, pp. 211-219, 1994.

[19] Girosi, F., M. Jones and T. Poggio, "Priors, stabilizers and basis functions: from regularization to radial, tensor and additive splines," A.I. Memo 1430, Massachusetts Institute of Technology, 1994.

[20] Gyllenberg, H. G., "Development of reference systems for automatic identification of clinical isolates of bacteria," *Archivum Immunologiae et Therapiae Experimentalis*, vol. 24, pp. 1-19, 1976.

[21] Gyllenberg, H. G., "Continuous cumulation of identification matrices," *Helsingin Ylioposton Mikrobiologian Laitoksen Julkaisuja*, vol. 20, 1981.

[22] Hartman, E. and J. D. Keeler, "Predicting the future: advantages of semilocal units," *Neural Computation*, vol. 3, no. 4, pp. 566-578, 1991.

[23] Haykin, S., *Adaptive Filter Theory*, 2nd ed. Englewood Clifs, NJ: Prentice Hall, 1991.

[24] Haykin, S., *Neural Networks, a Comprehensive Foundation*, New York: IEEE Computer Society Press, 1994.

[25] Hebb, D. O., *The Organization of Behavior*, New York: John Wiley & Sons, inc., 1949.

[26] Hecht-Nielsen, R., "Applications of counterpropagation networks," *Neural Networks*, vol. 1, pp. 131-139, 1988.

[27] Holland, J. H., *Adaptation in Natural and Artificial Systems*, 2nd ed. The MIT Press, 1992.

[28] Hornik, K., M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359-366, 1989.

[29] Hudak, M. J., "RCE classifiers: theory and practice," *Cybernetics and Systems*, vol. 23, pp. 483-515, 1992.

[30] Jaeckel, L. A., "An alternative design for sparse distributed memory," Tech. Rep. 89.28 RIACS, NASA Ames Research Center, Moffet Field, CA, 1989.

[31] Joglekar, U. D., "Learning to read aloud: a neural network approach using sparse distributed memory," Tech. Rep. 89.27 RIACS, NASA Ames Research Center, Moffet Field, CA, 1989.

[32] Jordan, M. I. and R. A. Jacobs, "Hierarchies of adaptive experts," in *Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson and R. P. Lippmann Eds. Denver, CO, USA, 1991, pp. 958-992.

[33] Kadirkamanathan, V. and M. Niranjan, "A function estimation approach to sequential learning with neural networks," *Neural Computation*, vol. 5, no. 6, pp. 954-975, 1993.

[34] Kanerva, P., *Sparse Distributed Memory*, Cambridge, MA: MIT press, 1988.

[35] Kanerva, P., "Efficient packing of patterns in sparse distributed memory by selective weighting of input bits," in *Proceedings of the 1991 International Conference Artificial Neural Networks*, T. Kohonen, et al. Eds. Espoo, Finland, 1991, vol. 1, pp. 279-284.

[36] Kanerva, P., "Sparse distributed memory and related models," *Associative Neural Memories; Theory and Implementations*, M. H. Hassoun Ed. Oxford, UK: Oxford Univ. Press, 1993.

[37] Kohonen, T., "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, pp. 59-69, 1982.

[38] Kohonen, T., *Self-Organization and Associative Memory*, 2nd ed. Berlin: Springer-Verlag, 1988.

[39] Kohonen, T., "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464-1480, 1990.

[40] Lampinen, J., "Distortion tolerant pattern recognition using invariant transformations and hierarchical SOFM clustering," in *Artificial Neural Networks. Proceedings of the 1991 International Conference. ICANN-91*, Espoo, Finland, 1991, vol. 1, pp. 99-104.

[41] Lippmann, R. P., "An Introduction to computing with neural nets," *IEEE Acoustics, Speech, and Signal Processing Magazine*, vol. 4, pp. 4-22, 1987.

[42] Lutterell, S. P., "Hierarchical vector quantization," *IEE Proceedings (London)*, vol. 136, Part I, pp. 405-413, 1989.

[43] Mao, J. and A. K. Jain, "Regularization techniques in artificial neural networks," in *World Congress on Neural Networks*, Portland, OR, USA, 1993, vol. 4, pp. 75-79.

[44] Mitchie, D. and R. Chambers, "BOXES: An experiment in adaptive control," *Machine Intelligence 2*, E. Dale and D. Mitchie Eds. Edinburg: Oliver and Boyd, pp. 137-152, 1968.

[45] Moody, J., "Fast learning in multi-resolution hierarchies," in *Neural Information Processing Systems 1*, D. Touretzky Ed., Denver, CO, 1988, pp. 29-39.

[46] Moody, J. and C. J. Darken, "Learning with localized receptive fields," in *Proceedings of the 1988 Connectionist Summer School*, D. Touretzky, Hinton and Sejnowski Eds. 1988,

[47] Moody, J. and C. J. Darken, "Fast learning in networks of locally-tuned processing units," *Neural Computation*, vol. 1, pp. 281-294, 1989.

[48] Neural Networks Group, "Design and implementation of a recognition accelerator," To appear in 1993 Proceedings of the Canadian Conference on Very Large Scale Integration. Intel Corporation, Santa Clara, CA, USA, 1993.

[49] Nordström, T., "Designing parallel computers for self organizing maps," in *DSA-92, Fourth Swedish Workshop on Computer System Architecture*, Linköping, Sweden, 1992,

[50] Nordström, T., "Hardware for sparse distributed memory simulations," *to be submitted*, 1995.

[51] Nordström, T., "On-line spatially localized learning systems, part II - parallel computer implementation," *to be submitted (Also available as Res. Rep. TULEA 1995:2, Luleå University of Technology, Sweden)*, 1995.

[52] Nordström, T. and B. Svensson, "Using and designing massively parallel computers for artificial neural networks," *Journal of Parallel and Distributed Computing*, vol. 14, no. 3, pp. 260-285, 1992.

[53] Nowlan, S. J., "Maximum likelihood competitive learning," in *Neural Information Processing Systems 2*, D. Touretzky Ed., Denver, CO, 1989, pp. 574-582.

[54] Park, J. and I. W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural Computation*, vol. 3, no. 2, pp. 246-257, 1991.

[55] Park, J. and I. W. Sandberg, "Nonlinear approximations using elliptic basis function networks," *Circuits Systems Signal Processing*, vol. 13, no. 1, pp. 99-113, 1993.

[56] Parzen, E., "On estimation of a probability density function and mode," *The Annals of Mathematical Statistics*, vol. 33, no. 1962, pp. 1065-1076, 1962.

[57] Platt, J. C., "Learning by combining memorization and gradient descent," in *Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S. Touretzky Eds. Denver, CO, USA, 1990, pp. 714-720.

[58] Poggio, T. and F. Girosi, "Networks for Approximation and Learning," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1481-1497, 1990.

[59] Poggio, T. and F. Girosi, "A theory of networks for approximation and learning," A.I. Memo 1140 (first released 1991), Massachusetts Institute of Technology, 1994.

[60] Pohja, S. and K. Kaski, "Kanerva's sparse distributed memory with multiple Hamming thresholds," Tech. Rep. 92.10, RIACS, NASA Ames Research Center, 1992.

[61] Powell, M. J. D., "Radial basis functions for multivariable interpolation: a review," in *IMA Conference on Algorithms for the Approximation of Functions and Data*, RMCS, Shrivenham, UK, 1985, pp. 143-167.

[62] Powell, M. J. D., "Radial basis function approximations to polynomials," in *Numerical Analysis 1987*, Dundee, UK, 1988, pp. 223-241.

[63] Prager, R. W., T. J. W. Clarke and F. Fallside, "The modified Kanerva model: results for real time word recognition," in *First IEE International Conference on Artificial Neural Networks*, London, UK, 1989, pp. 105.

[64] Prager, R. W. and F. Fallside, "The modified Kanerva model for automatic speech recognition," *Computer Speech and Language*, vol. 3, pp. 61-81, 1989.

[65] Reilly, D. L., L. N. Cooper and C. Elbaum, "A neural model for category learning," *Biological Cybernetics*, vol. 45, pp. 35-41, 1982.

[66] Richard, M. D. and R. P. Lippmann, "Neural network classifiers estimate Bayesian a posteriori probabilities," *Neural Computation*, vol. 3, no. 4, pp. 461-483, 1991.

[67] Rogers, D., "Predicting weather using a genetic memory: a combination of Kanerva's sparse distributed memory with Holland's genetic algorithms," in *Neural Information Processing Systems 2*, D. Touretzky Ed., Denver, CO, 1989, pp. 455-464.

[68] Rosenblatt, F., "The Perceptron: a probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386-408, 1958.

[69] Rumelhart, D. E. and J. L. McClelland, *Parallel Distributed Processing; Explorations in the Microstructure of Cognition*, vol. I and II Cambridge: MIT Press, 1986.

[70] Saarinen, J., S. Pohja and K. Kaski, "Self-organization with Kanerva's sparse distributed memory," in *Artificial Neural Networks. Proceedings of the 1991 International Conference. ICANN-91*, T. Kohonen, et al. Eds. Espoo, Finland, 1991, vol. 1, pp. 285-290.

[71] Saha, A., C.-L. Wu and D.-S. Tang, "Approximation, dimension reduction, and nonconvex optimization using linear superpositions of Gaussians," *IEEE Transactions on Computers*, vol. 42, no. 10, pp. 1222-1233, 1993.

[72] Simard, P., Y. L. Cun and J. Denker, "Efficient pattern recognition using a new transformation distance," in *Neural Information Processing Systems 5*, C. L. Giles, S. J. Hanson and J. D. Cowad Eds. Denver, CO, USA, 1992, pp. 50-58.

[73] Simard, P., B. Victorri, Y. L. Cun and J. Denker, "Tangent prop – a formalism for specifying selected invariances in an adaptive network," in *Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson and R. P. Lippmann Eds. Denver, CO, USA, 1991, pp. 895-903.

[74] Smith, D., M. Shetti, M. Harward, W. Bean, R. Pawate and G. Doddington, "A VLSI implementation of the Nestor RCE neural network," *Texas Instruments Technical Journal*, vol. 7, no. 6, pp. 34-41, 1990.

[75] Specht, D. F., "Probabilistic Neural Networks," *Neural Networks*, vol. 3, no. 1, pp. 109-118, 1990.

[76] Sutton, R. S. and S. D. Whitehead, "Online learning with random representations," *To appear*, 1993.

[77] Svensson, B. and T. Nordström, "Execution of neural network algorithms on an array of bit-serial processors," in *10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*, Atlantic City, NJ, USA, 1990, vol. II, pp. 501-505.

[78] Svensson, B., T. Nordström, K. Nilsson and P.-A. Wiberg, "Towards modular, massively parallel neural computer," *Connectionism in a Broad Perspective: Selected Papers from the Swedish Conference on Connectionism - 1992*, L. F. Niklasson and M. B. Bodén Eds. Ellis Horwood, pp. 213-226, 1994.

[79] Tan, S. and Y. Yu, "On-line stable nonlinear modelling by structurally adaptive neural nets," Neuroprose: yuyi.online_rbf.ps.Z (yuyi@ee.nus.sg), 1993.

[80] Tikhonov, A. N. and V. Y. Arsenin, *Solutions of Ill-posed Problems*, Washington DC: W. H. Winston, 1977.

[81] von der Malsburg, C., "Self-organization of orientation sensitive cells in the striate cortex," *Kybernetik*, vol. 14, pp. 85-100, 1973.

[82] Wan, E. A., "Neural network classification: a Bayesian interpretation," *IEEE Transaction on Neural Networks*, vol. 1, no. 4, pp. 303-305, 1990.

[83] Wong, Y., "How Gaussian radial basis functions work," in *International Joint Conference on Neural Networks*, Seattle, WA, USA, 1991, pp. 133-138.

[84] Xu, L., A. Krzyzak and E. Oja, "Rival penalized competitive learning for clustering analysis, RBF net, and curve detection," *IEEE Transactions on Neural Networks*, vol. 4, no. 4, pp. 636-649, 1993.

On-Line Localized Learning Systems Part II – Parallel Computer Implementation

Tomas Nordström

Division of Computer Science & Engineering
Luleå University of Technology, Sweden
E-mail: tono@sm.luth.se

ABSTRACT

This is the second paper in a series of two analyzing localized learning systems (LLSs). Whereas the first paper concentrated on the model, this paper will study parallel computer implementations of this model. The LLS is a superclass of important and commonly used artificial neural network models with many useful properties. Besides performing classification and approximation tasks well, the LLS model has shown promising possibilities to run efficiently on parallel computers, a possibility which will be explored in this paper. After analyzing the number and type of computations required we make suggestions on how to implement the models on parallel computing hardware.

It is established that a mapping which combines two forms of parallelism (node and weight) is the preferred form of mapping. To support this "mixed parallelism" a very simple structure is found to be sufficient, consisting of a linear single instruction stream, multiple data streams (SIMD) array using broadcast communication. In addition most variations of LLSs require extra support for reduction operations. In this paper we suggest and analyze effective means to do reduction-sum and minimum (winner-take-all) operations.Three implementations of global-sum are identified and studied. It is found that a bit-serial tree of adders gives the best performance/size ratio. For the global-minimum operation a new bit-serial structure is proposed. This new min/max network has the advantage of not needing a global-or network as the standard bit-serial way of finding minimum does. This also results in a speed advantage in most cases.

1. Introduction

This paper is our second paper about on-line localized learning systems (LLSs). We have described and analyzed LLSs in a companion paper, Part I [29], and we assume the reader is familiar with that material. Here we will concentrate on the parallel computer implementation aspects of this model. LLSs can be seen as a "superclass" of artificial neural network (ANN) models. Some of the LLS variations are sometimes referred to as kernel or basis function networks. In essence the LLS model is a *feedforward* ANN forming an *expanded representation* (most nodes are in the hidden layer), having *local activity* (only a subset of nodes are causing significant output at a certain time), and *localized learning* (only active nodes are updated). In these papers we further restrict ourselves to variations that can be used in an *on-line* fashion (learning done incrementally and in real-time).

A number of ANN models can be found as variations of the LLS model. Therefore by implementing the LLS model a number of well known ANN models will also be implemented. Models found to be LLSs are generalized radial basis functions (GRBF) [33, 34], self-organizing feature maps (SOFM) and learning vector quantization (LVQ) [22], restricted Coulomb energy (RCE) [6, 17], probabilistic neural network (PNN) [41], sparse distributed memory (SDM) [21], and cerebellar model arithmetic computer (CMAC) [2]. All these models are commonly used and it is interesting to look for a common hardware platform for all these ANN models, and this is provided through the LLS concept. We hope to utilize the natural parallelism in the many nodes, and the use of local activation and learning, to be able to find efficient parallel computer implementations.

This paper is one paper in a series of papers [26, 27, 30, 42] studying the implementation of ANNs on parallel computers. In our earlier studies we have found that a surprisingly simple architecture is enough to get good performance on many ANN simulations. That is, we have found a linear array, with broadcast and support for multiplication, to be suitable for almost all the ANN models we have studied so far. This is also the starting point for this paper. However, it will be found that some extra supporting hardware can be useful if improved performance is required for LLS computations.

1.1 Types of architectures

There is always a choice between dedicated designs and general purpose designs. The highest performance is possible by dedicated design, but as long as the algorithmic details are not decided, it can be risky to make a special design only to find it implementing an outmoded algorithm. In other words, one often pays in flexibility for high performance. We have found that many times the linear array computing in SIMD (single instruction stream, multiple data streams) mode is a good trade-off between flexibility and performance. This style of computing is also called associative array processor [12]. It is interesting to note that the LLS model in some respects implements an associative memory, which might indicate a good fit to this style of computing.

In [23] S. Y. Kung discusses systolic implementations of ANN, mostly multilayer perceptrons (MLP) with back propagation (BP) learning. It is noted that 2-D arrays can only be

efficiently used if training data parallelism (batch processing) is utilized. For the on-line (per pattern) learning the “best” structure becomes a linear systolic array. In his design there is only nearest neighbor communication, whereas we allow broadcast as well.

In the next section we shortly restate the LLS algorithm and some of its variants. Then a general discussion on parallelism found in ANN calculation and how to map these calculations onto a parallel computer follows. Next there is a section with a detailed analysis of each step in the LLS algorithm.

We also discuss two areas with difficult implementation aspects, the calculation of the radial function and the reduction operations. This is followed by a section where we summarize the architectural support which LLSs need. After an overview of related work we conclude the paper.

2. Recapture of the main aspects of LLS

In this Section we briefly summarize the main aspects of the LLS model. As noted in the introduction the LLS is a *feedforward* ANN forming an *expanded representation* (most nodes in the hidden layer). The feed-forward phase can be visualized as in Figure 1a. In Figure 1b the data flow and data organization of the feedforward phase are shown.

The main characteristic of the model is the *local activity* (only a subset of nodes are active at the same time in the hidden layer), and the *localized learning* (only active nodes are updated). We are mainly interested in variations which allow training to take place after each new training data, that is the LLS is used in an *on-line* fashion.

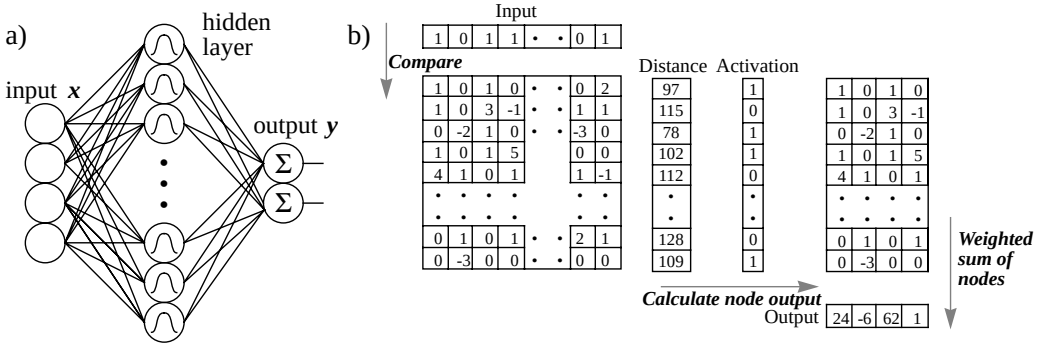

Figure 1 a) The LLS model is a feedforward neural network model with localized activity at the hidden nodes. b) The data flow and data organization of the LLS model (feedforward phase).

The feedforward phase for an LLS with M nodes and multiple outputs, can be written as

$$F_j(\boldsymbol{x}_p, \Theta) = \sum_{i \in A} w_{ij} \varphi(r_i), \tag{1}$$

where $\varphi(r_i)$ is the ith node output, A is the set of active nodes, $\boldsymbol{x}_p$ is the input, and w_{ij} is the weight connecting node i with output j. The node output will depend on the distance measurement used to calculate the distance r_i to some centers (templates) $\boldsymbol{c}_i$, size and form of the receptive field S_i, and type of kernel function φ. A node with a significant output is called a *selected* or *active node*, that is, a node is in the set of active nodes A if $\varphi(r_i) \geq \alpha$, with $\alpha \geq 0$ being a constant. The free parameters to be set, usually by training, are $\Theta = \{\boldsymbol{w}_i, \boldsymbol{c}_i, S_i\}_{i=1}^{M}$. A number of variations exists and the variations discussed in this paper are shown in Table 1. To exemplify we will use the Euclidean distance, a Gaussian kernel function $\varphi(r_i) = \exp(-r_i^2)$, and a general receptive field, that is, $r_i^2 = \|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S = (\boldsymbol{x} - \boldsymbol{c}_i)^T S_i (\boldsymbol{x} - \boldsymbol{c}_i)$, where S_i can be recognized as the inverse of the covariance matrix.

The equations for training the free parameters of Eq. (1), using a *gradient descent* method, were developed in Part I [29]. We found that the hyperbolic receptive field, that is $S_i = \text{diag}\,[s_1, \ldots, s_d]_i$, can serve as a reasonable example (the most complex receptive field still useful for the parallel computer implementations). Using $\boldsymbol{F}(\boldsymbol{x}_p, \Theta) = [F_1(\boldsymbol{x}_p, \Theta), \ldots, F_m(\boldsymbol{x}_p, \Theta)]^T$ as the network output vector with m elements,

we can write the network error vector as $\boldsymbol{e}_p = \boldsymbol{F}(\boldsymbol{x}_p, \Theta) - \boldsymbol{y}_p$. Below we have the update equations for a per pattern gradient descent using a least squares error:

$$\boldsymbol{w}_i(t+1) = \boldsymbol{w}_i(t) - 2\eta_w \boldsymbol{e}_p \varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S), \quad (2)$$

$$\boldsymbol{c}_i(t+1) = \boldsymbol{c}_i(t) + 4\eta_c \varepsilon_{ip} \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S) S_i (\boldsymbol{x}_p - \boldsymbol{c}_i), \quad (3)$$

$$S_i(t+1) = S_i(t) - 2\eta_S \varepsilon_{ip} \varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S) (diag[(\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})^2, \ldots, (\boldsymbol{x}_{pd} - \boldsymbol{c}_{id})^2]), \quad (4)$$

for $i \in A$ and $\varepsilon_{ip} = \sum_{j=1}^{m} e_{pj} w_{ij}$.

Not all variations update all these free parameters, and still others do not use the gradient update at all. Another common way to update the kernel centers is to use *competitive learning* which, in a refined form (rival penalized competitive learning [47]), becomes:

$$\boldsymbol{c}_i(t+1) = \boldsymbol{c}_i(t) + u_i (\boldsymbol{x}_p - \boldsymbol{c}_i), \quad (5)$$

$$\text{where } u_i = \begin{cases} \eta_k & \text{if } i = k \text{ such that } \gamma_k \|\boldsymbol{x}_p - \boldsymbol{c}_k\|_2 = \min_j \gamma_j \|\boldsymbol{x}_p - \boldsymbol{c}_j\|_2, \\ -\eta_r & \text{if } i = r \text{ such that } \gamma_r \|\boldsymbol{x}_p - \boldsymbol{c}_r\|_2 = \min_{j \neq k} \gamma_j \|\boldsymbol{x}_p - \boldsymbol{c}_j\|_2, \\ 0 & \text{otherwise,} \end{cases} \quad (6)$$

for $i = 1, \ldots, M$ and $0 \leq \eta_k, \eta_r \leq 1$. Usually $\eta_k(t) \gg \eta_r(t)$. Moreover, $\gamma_j = n_j / (\sum_{i=1}^{M} n_i)$, which of course is equal to n_j / N, and n_i is the cumulative number of occurrences of $u_i = \eta_k$. That is to say that the node closest to the input (node k) is moved towards the input and that the second best node (the runner up) r is moved away. To involve all nodes the distances are weighted with the number of inputs assigned to a certain node.

Many other variations exist both on how to initiate the parameters and how to adapt them. The variations used in LLSs are shown in Table 1. We refer to Part I for a more thorough discussion on the variations found in Table 1. This paper assumes fixed-point (integer) numbers unless otherwise stated. Variations with floating-point (real) or Boolean input will not be discussed.

Input	Distance	Receptive field	φ	$\boldsymbol{c}_i$ initiation	$\Delta\boldsymbol{c}$	Δw	ΔS	Output
R	L_1 (cityblock)	1	exp	Random	Fixed	Pseudo-inverse	Fixed	R
I	L_2 (Euclidean)	sI	Radial	Uniform	Gradient	Gradient	Gradient	I
B	L_∞	$s_i I$	Threshold logic unit	Subset of data	Competitive learning	Occurrence (Hebb)	RCE	B
	Dot product	$\mathrm{diag}\,[s_j]$	Min/Max	All data	+ Topology			
	Hamming distance	$\mathrm{diag}\,[s_j]_i$			Incremental addition			
		S_{ij}			Genetic algorithm			
		Hierarchical						
		Sample/Hash						

Table 1 Variations of the LLS model. A large family of LLS variations is found by varying such things as input and output spaces (Real, Integer or Boolean); distances, e.g., cityblock (L1), or Euclidean (L2); receptive field (e.g., sphere $S = s_i I$ or elliptic $S = \mathrm{diag}\,[s_j]_i$); kernel function (exponential, threshold, min/max etc.); and how to initiate and adapt the free parameters.

Some of the LLS variants shown in Table 1 are not analyzed further, namely the variants with *genetic algorithm*, *sample/hash*, or *incremental addition of nodes* (structural adaptation), as they are not main stream LLS models. Due to the non-local nature of the *pseudoinversion* method, and its computational demand making it hard to use in on-line situations, the pseudoinversion method to set $\boldsymbol{w}_i$ is not studied either.

3. Types of parallelism

Because of the regular structure and repeated calculations over a large number of nodes, the natural computer implementation will use a highly or massively parallel architecture (these terms are defined in [30]). Before the best mapping can be found it is important to identify the kind of parallelism found in the algorithm.

The different types of parallelism typically found in ANN algorithms [30] are:

Training session parallelism
Training example parallelism
Layer and Forward-Backward parallelism
Node (neuron) parallelism
Weight (synapse) parallelism
Bit parallelism

The training session parallelism uses the parallelism in sessions with different training parameters. The training example parallelism exploits the parallelism in the large number of training examples available, this will however make the algorithm batch (off-line) oriented. One example is the pseudoinverse method to determine w_i. The layering (or pipelining) parallelism is not large and is seldom used. Both the parallelism in nodes (the many neurons) and weights (the many synapses per neuron) are commonly used. The bit parallelism is often taken for granted but becomes visible for bit-serial approaches. The greatest parallelism is found in training example parallelism, node (neuron) parallelism, and weight (synapse) parallelism, therefore these three are the most interesting to exploit for parallel computer implementations. As this paper focuses on on-line methods, we conclude that the node and weight parallelism are the most attractive forms of parallelism.

If we expand Eq. (1) we get, using a Euclidean distance without weighting,

$$y_j = \sum_{i \in A} w_{ij} \varphi(r_i), \tag{7}$$

where d is the number of inputs, $r_i = \sum_{k=1}^{d} (x_k - c_{ik})^2$, and $\text{Node}_i \in A$ if $\varphi(r_i) \geq \alpha$, with $\alpha \geq 0$ being a constant. This condition for activity can usually be written as $r_i \leq \rho$, with $\rho \geq 0$ being a constant. Thus, each and every node needs to calculate this distance to find the active set. A node parallel solution for this calculation uses M nodes operating in parallel, while a weight parallel solution uses d nodes. That is, the computational time for the distance calculation is proportional to d for a node parallel solution, and M for a weight parallel solution. As we assume that we have an expanded representation we know that $M > d$, and we can conclude that the node parallel solution is the most efficient one for the distance calculating phase. Using the same reasoning for the output layer, that is, Eq. (7), and noting that $m < M$, we find the weight parallelism potentially faster. This, however, will be depending on the support available to combine (sum) the weight parallel calculations. Later, in Section 4.2, we will discuss how to implement an global-sum and other reduction operations.

A combination of the weight and the node parallelism while calculating a single layer is of course also possible. This combination is in fact the maximally parallel solution in the on-

line case. Inspecting Table 2 we find that a typical feedforward phase computes subtraction, squaring, weighting with s_{kj}, the sum $\sum_{k=1}^{d} s_{kj}(x_k - c_{ik})^2$, weighting with w_{ij}, and the final sum $\sum_{i=1}^{M} w_{ij}\varphi$. For the worst case scenario, with $|A| = M$, a maximally parallel solution for the feedforward phase only uses $\log M + \log d + c_1$ steps, where c_1 is a small constant (≈5). Similarly, updating all free parameters using a gradient method can be done in $c_w + \log m + c_c + c_S$ steps where c_w, c_c, and c_S are small constants (≈4, ≈6, and ≈4, respectively). However, this maximally parallel solution is unrealistic due to the hardware resources it needs. For the feedforward phase alone we need dM multipliers, M adder trees of length d, m adder trees of length M, and mM multipliers. The hardware would probably also need to be specialized towards certain network sizes.

4. Detailed analysis of the LLS algorithm

Our analysis of the LLS algorithm will be based on an implementation which uses node parallelism for the hidden nodes and weight parallelism for the output nodes. For computations we will use addition and multiplication as the basic operations and relate other operations to them, e.g., we can assume that a subtraction and the calculation of an absolute value to take the same time as an addition. In Table 2 most of the variations for the feedforward phase are analyzed. This is followed by an analysis of the updating phase summarized in Table 3. We will continue to use d as the number of inputs, M as the number of nodes, and m as the number of outputs.

For the different variations of distance measurements we see that the dot product is the fastest, unless normalizations are needed. For the Hamming distance (HD) it is possible to make use of the fact that all inputs are binary and use an exclusive-or operation followed by counting the bits in the result. While using the Mahalanobis distance (MD) we implicitly use an, L_2 distance. The shown receptive fields are special cases of the MD as long as the L_2 distance is used. The large number of operations for the MD, especially for long input vectors, makes it less interesting for high performance applications.

As a kernel function the Gaussian is much used, but the calculation of it needs special consideration. Ways to calculate it, or approximate it, will be discussed later. To support LLS variations using winner-take-all it is required to find the minimum or maximum across a node parallel mapping. Likewise, a weight parallel mapping will require a global sum across the same nodes. Different ways to support these two forms of reduction operations (marked with gray in Table 2) will be discussed in Section 4.2.

The error can either be calculated in the host/controller or in the processor array if the network output vector $\boldsymbol{F}(\boldsymbol{x}_p, \Theta)$ and the target vector $\boldsymbol{y}$ are broadcast.

Variation Type	Variation	Result	Operation	Additions	Mult.	Other ops.
Distance	L_1	δ_k	$\lvert x_k - c_{ik}\rvert$	$2d$		
	L_2	δ_k	$(x_k - c_{ik})^2$	d	d	
	L_∞	δ_k	$\min_k(\lvert x_k - c_{ik}\rvert)$	$2d$		$\log d$, or d
	Dot product	δ_k	$x_k c_{ik}$		d	
	HD	δ_k	" $(x_k - c_{ik})^2$ "			exclusive-or
Mahalanobis distance (MD)	S_{ij}	r_i^2	$(x_k - c_{ik})^T S (x_k - c_{ik})$	$d^2 + d - 1$	$d^2 + d$	
Receptive field	1	$r_i, (r_i^2)$	$\sum_k \delta_k$	$d - 1$		
	sI	$r_i, (r_i^2)$	$s\sum_k \delta_k$	$d - 1$	1	
	$\mathrm{diag}\,[s_k]_i$	$r_i, (r_i^2)$	$\sum_k s_k \delta_k$	$d - 1$	d	
Kernel function	exp	$\varphi(\ )$	$\exp(-r_i^2)$			2-10
	TLU	index set A	A = all i where $r_i \le \rho$			1
	Min/Max	index k, (k_2)	$\min_j r_j, (\mathrm{second}_j\, r_j)$			q, $\log\lvert A\rvert$
Output and error	exp	$\boldsymbol{F}(\boldsymbol{x}, \Theta)$	$\sum_i w_i \varphi(\ )$	$m\log\lvert A\rvert$	m	
	TLU	$\boldsymbol{F}(\boldsymbol{x}, \Theta)$	$\sum_{i \in A} w_i \varphi(\ )$	$m\log\lvert A\rvert$	m	
	•	$\boldsymbol{e}_p$	$\boldsymbol{F}(\boldsymbol{x}, \Theta) - \boldsymbol{y}$	m		

Table 2 The number of operations needed for the feedforward phase of the LLS algorithms for most of the variations given in Table 1. d is the number of inputs, $|A|$ the number of active nodes, m is the number of outputs, and q the number of bits used to represent node activity. For all operations, except for the ones in the gray rows, the number of operations is found by multiplying the shown figure by the number of nodes M. That is, this is the number of steps needed for a node parallel solution.

We also need to analyze the means of communication that these computations need. For most computations the only communication needed is broadcast, that is, a *one to many communication* (broadcasting of the input vector $\boldsymbol{x}_p$, the radius ρ, and the error $\boldsymbol{e}_p$). The "inverse" of broadcasting is a reduction operation needing *many to one communication*. This operation is found while calculating the distance L_∞, the output $\boldsymbol{F}(\boldsymbol{x}, \Theta)$, or finding the winning node (minimum/maximum). The type of operation used for the combination of the values will determine the type of reduction, e.g., global-or, minimum, or global-sum. We note that in a SIMD computer there already exists a broadcast from the controller to the processing elements (PEs) namely the instruction.

In Table 3 we show the number of operations which are needed for the updating of the free parameters. Note that the node output $\varphi(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S)$ is already calculated in the feedforward process, and that the factor 2 or 4 can be incorporated into the learning rates η. While updating S_i we try to reuse as much of the previously calculated partial results as possible, that is, $\varepsilon_{ip}\varphi'(\|\boldsymbol{x}_p - \boldsymbol{c}_i\|_S^2)$, $(\boldsymbol{x}_p - \boldsymbol{c}_i)^T (\boldsymbol{x}_p - \boldsymbol{c}_i)$, and $(x_{pj} - c_{ij})$ are reused. The number of opera-

tions needed while updating $\Delta \boldsymbol{c}$ and ΔS will depend on the chosen receptive field. We show the numbers for $s_i I$, diag $[s_i]$, and S_{ij}. For the global versions, that is, sI and diag $[s_j]$, the $S(t+1)$ becomes $S(t) + \sum \Delta S$ with a correspondingly larger number of operations. Note that this summation is a global-sum across the PEs.

Comparing methods to calculate $\Delta \boldsymbol{c}$ in Table 3 we see that a gradient method using diag $[s_i]$ as receptive field has a similar time consumption as the competitive learning methods. For the gradient method more nodes are updated at the same time. For a sequential computer we can utilize the fact that only one or two nodes should be updated, but that is not the case for a parallel computer (at least not on the mapping suggested). As the methods are very different (e.g., supervised versus unsupervised) it is not certain that the lesser number of nodes updated will lead to worse performance (it could even lead to better performance in some situations). The addition of a topological constraint leads to a larger number of nodes being updated for each training pattern, this without taking a significantly longer time on our suggested architecture.

To find the neighboring nodes for LLS variations that use topological constraints, the fastest way is often to broadcast the winning nodes' "topological position" and let all nodes calculate their respective "topological distance" to the winning node. About three operations per topological dimension should be sufficient.

We can also note that, as for the feedforward phase, the Mahalanobis distance (general S_{ij}) is not advantageous for long input vectors because of the $O(d^2)$ operations needed.

Variation Type	Variation	Result	Operation	Additions	Mult.	Comments and other ops.
$\Delta \boldsymbol{w}$	Gradient	$\Delta \boldsymbol{w}$	$\eta_w \boldsymbol{e}_p \varphi(\ )$		$1 + m$	
	Occurrence	$\Delta \boldsymbol{w}$	$\boldsymbol{w}_k(t) + \boldsymbol{y}$	m		Normalization is needed
	•	$\boldsymbol{w}_i(t+1)$	$\boldsymbol{w}_i(t) + \Delta \boldsymbol{w}$	m		
$\Delta \boldsymbol{c}$	Gradient (L_2, exp)	ε_{ip}	$\sum_{j=1}^{m} e_{pj} w_{ij}$	$m-1$	m	
			$\varphi'(-r_i^2)$			2-10 ops.
		$\partial r^2 / (\partial c)$	$S_i (\boldsymbol{x}_p - \boldsymbol{c}_i)$	$\bar{\ }, \bar{\ }, d^2 - d$	$1, d, d^2$	
		$\Delta \boldsymbol{c}$	$\eta_c \varepsilon_{ip} \varphi'(-r_i^2) \dfrac{\partial r^2}{\partial c}$		$1 + d$	
	CL	$\Delta \boldsymbol{c}_k$	$\eta_k (\boldsymbol{x} - \boldsymbol{c}_k)$, for index k or all $k \in A$		d	
	RPCL	$\Delta \boldsymbol{c}_k$	$\eta_k (\boldsymbol{x} - \boldsymbol{c}_k)$		d	Actually local to one or two nodes
		$\Delta \boldsymbol{c}_r$	$-\eta_r (\boldsymbol{x} - \boldsymbol{c}_r)$		d	
		n_k	$n_k + 1$	1		
	CL + topology.	$\Delta \boldsymbol{c}_k$	$\eta_k N_k (\boldsymbol{x} - \boldsymbol{c}_k)$, where N_k is the topological distance to k		$2d$	Need to calculate the topological distance
	•	$\boldsymbol{c}_i(t+1)$	$\boldsymbol{c}_i(t) + \Delta \boldsymbol{c}$	d		
ΔS	Gradient: L_2, exp, $s_i I$	$\partial r^2 / (\partial S)$	$(\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})^T (\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})$	–	–	
		ΔS	$\eta_S \varepsilon_{ip} \varphi'(-r_i^2) \dfrac{\partial r^2}{\partial S}$		2	
		$S_i(t+1)$	$S_i(t) + \Delta S$	1		
	Gradient: L_2, exp, $\mathrm{diag}\,[s_j]_i$	$\partial r^2 / (\partial S)$	$(\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})^2$		d	
		ΔS	$\eta_S \varepsilon_{ip} \varphi'(-r_i^2) \dfrac{\partial r^2}{\partial S}$		$1 + d$	
		$S_i(t+1)$	$S_i(t) + \Delta S$	d		
	Gradient: L_2, exp, S_{ij}	$\partial r^2 / (\partial S)$	$(\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})\ (\boldsymbol{x}_{p1} - \boldsymbol{c}_{i1})^T$		d^2	
		ΔS	$\eta_S \varepsilon_{ip} \varphi'(-r_i^2) \dfrac{\partial r^2}{\partial S}$		$1 + d^2$	
		$S_i(t+1)$	$S_i(t) + \Delta S$	d^2		
	RPCL	n_k	$n_k + 1$	1		
		$s_i(t+1)$	$n_k (1/N)$		1	global $1/N$

Table 3 The number of operations needed for updating the free parameters of the LLS algorithms. For all operations the number of operations is found by multiplying the figures by $|A|$.

Timing example

Using a Euclidean distance, a hyperbolic receptive field ($\mathrm{diag}\,[s_k]_i$), a Gaussian kernel, and gradient learning in Eq. (1)-(4), we can compute the per pattern update time (using a node parallel mapping onto a linear array with M processors) as

$$\begin{aligned} T_{tot} &= dT_{add} + dT_{mul} + (d-1)\,T_{add} + dT_{mul} + T_{\varphi} + mT_{mul} + mT_{GS} + mT_{add} \\ &+ (1+m)\,T_{mul} + mT_{add} \\ &+ (m-1)\,T_{add} + mT_{mul} + T_{\varphi'} + dT_{mul} + (1+d)\,T_{mul} + dT_{add} \\ &+ dT_{mul} + (1+d)\,T_{mul} + dT_{add} \\ &= (4d+3m-2)\,T_{add} + (6d+3m+3)\,T_{mul} + T_{\varphi} + mT_{GS} + T_{\varphi'} \end{aligned} \quad \textbf{(8)}$$

The times T_{add} and T_{mul} are the lengths of time consumed by an addition and a multiplication, respectively. The times T_{φ} and $T_{\varphi'}$ are the times needed to compute the radial function and its derivative. The time T_{GS} is the time needed to calculate a global-sum across the linear array. The trivial solution using only local (nearest neighbor) communication will calculate this global-sum in $|A|$ (i.e., in worst case M) steps. This is clearly undesirable, as we usually have an expanded representation where the number of kernels M is much larger than both the input vector length d and the output vector length m. If an adder tree is used, as described later, we can assume for the worst case (cf. end of Section 4.2), $T_{GS} = (\log M)\,T_{add}$.

For a bit-parallel computer two reasonable assumptions are that $T_{mul} = 2T_{add}$ and $T_{\varphi} + T_{\varphi'} = 10T_{add}$. Altogether these assumptions lead to a per pattern total time of

$$T'_{tot} = (16d + 9m + \log M + 14)\,T_{add}. \quad \textbf{(9)}$$

With mixed parallelism and support for reduction operations the term dependent on M becomes $\log M$ and the critical term for T'_{tot} is the term $16d$.

Most LLS variations are used with networks using maximally $d = 64$ inputs, $m = 64$ outputs, and $M = 2048$ nodes. This maximum gives a maximum $T'_{tot} = 1625T_{add}$. If a 25MHz clock is used and an add takes one clock cycle this maximally sized network could be updated at rate of 15000 updates/s. This is sufficient for many real-time applications, and is very high in situations involving humans.

By introducing the concept of virtual PEs it is possible to reduce the hardware required with the sacrifice of update speed. By virtual PEs it is meant that each PE is simulating many virtual PEs (giving the impression of having more PEs than what is actually available). The introduction of virtual PEs demands little or no extra support from the hardware, but requires more memory to be available for each PE. For a low level of virtualization (2-8 times) the update rate can be expected to be reduced in proportion to the level of virtualization.

4.1 Radial functions

It is well known how to implement the addition and multiplication operations [10, 18, 43]. However, for radial functions like the Gaussian, it is not as obvious which methods are available and which one should be preferred. This is especially true if reduced precision can be allowed as a way to achieve higher speed. A standard way to calculate an exponential

function is to use its truncated power-series as an approximation [18]. If we truncate all but one term we get an approximation in the form:

$$\varphi_i(r_i) = \exp(-r_i^2) \approx \begin{cases} \left(1 - \left(\frac{1}{q} r_i\right)\right)^2 & \text{if} \quad r_i < q \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

where $r_i^2 = \|\boldsymbol{x} - \boldsymbol{c}_i\|_S$ and $q = 2.67$ for best fit to a Gaussian [32]. If even lower precision is acceptable a version with $q = 2$ should be considered, as it makes Eq. (10) easier to calculate.

The cut-off at $r_i < q$ makes the kernel function truly local, which is not true for the exponential function we try to approximate. In a sense, a Gaussian kernel function does not conform to the LLS concept of locality, due to its infinite tail. Usually this can be resolved by introducing a cut-off for the implemented kernel function. This "reintroduction" of locality is especially needed if the method to do global-sum depends on the fact that there are few active nodes.

If trigonometric calculations already are implemented, or if they are calculated faster than the exponential, then the kernel function

$$\varphi_i(r_i) = \begin{cases} (1 + \cos r_i)/2 & \text{if} \quad r_i < \pi \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

could be considered. There are actually many other types of kernel functions that could be considered, both approximations that are close to a Gaussian, like the ones above, and others like a triangle function. Wong [46] actually suggests that the Gaussian kernel function is "too smooth" for some problems where there are high frequency sections to be mapped, and that a first order spline like the triangle function is better.

Another method to quickly compute an arbitrary function is to use a table lookup. This however needs the PEs to be autonomous enough to generate local addresses. Most SIMD computers do not have local address modification available for this kind of operation. For a discussion on autonomy of PEs in SIMD computers see [8]. The CNS computer from ICSI [4] will use a table lookup to calculate the sigmoid of the multilayer perceptrons (MLP), which is a similar problem to calculating the exponential. This method will need memory for the table in each PE, making strong restrictions on how long tables can be, and therefore on the precision. (A precision of 8 bits is used in [4].)

In [30] various approximations of the sigmoid for MLPs are discussed in greater detail, and many of the methods there can be applied to the exponential as well. It should for instance be easy to generate a piecewise linear approximation.

4.2 Reduction operations

For both node parallel, weight parallel, and maximally parallel implementations it is interesting to find fast ways of doing reduction operations. In the following sections we will study methods to calculate a global-sum and methods to find a global minimum (which is needed for "winner take all" algorithms).

Global-sum

There are many possible ways to add M numbers together in parallel, but the most efficient way is to use a (binary) tree structure. Here we study three variants: the *fully parallel* adder tree which uses parallel adders in all stages; the *partly parallel* adder tree which is sequential over the inputs, but adds the bits in each bit-slice with parallel adders; and a completely *bit-serial* adder tree.

- The *fully parallel* method is the most complex, using a complete adder for each crossing in the tree. The reduction will be completed in $\log M$ pipelined steps, needing $M-1$ q-bit (and larger) adders, where q is the number of bits in the numbers to be summed. We also need $(M-1)/2$ pipeline registers. The critical operation is a $(q+\log M-1)$-bit adder. This structure can easily be extended to other reduction operations by modifying the arithmetic/logic unit (ALU). In [39] Raynolds et al. described a general reduction network, similar to the one described above, capable of all important operations including minimum.

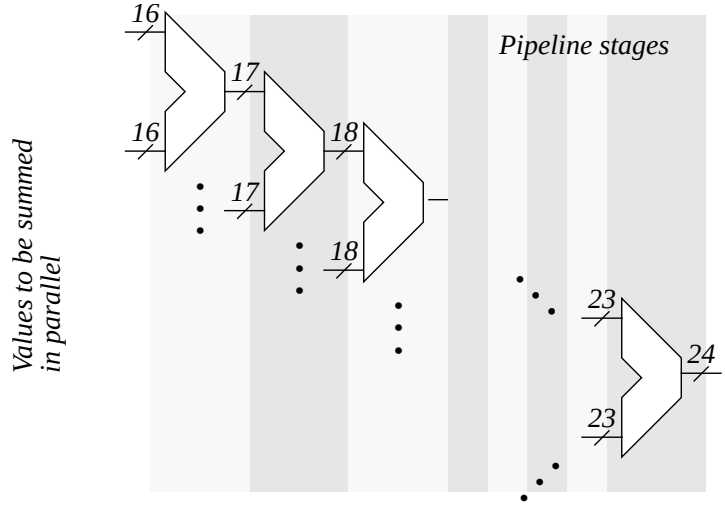

Figure 2 Part of a fully parallel adder tree to sum $M = 256$ numbers with maximally $q = 16$ bits each. Not shown are the pipeline registers between each layer of adders.

- The *partly parallel* adder tree forms an accumulated sum by adding up bit-slices from all the numbers, sent to the adder tree bit by bit. After $q + \log(M/p)$ steps the sum is complete. This solution needs M/p "p-bit to $\log p$-bit sum" arithmetic units, $\log(M/p) - 1$ adders with a length starting from $\log p$, and an accumulator with the length $q + \log M$. There are also $2(M/p)$ pipeline registers. The critical operation is the $(\log M + 1)$-bit and $(q + \log M)$-bit addition. The structure is inflexible and cannot be extended to other reduction operations.

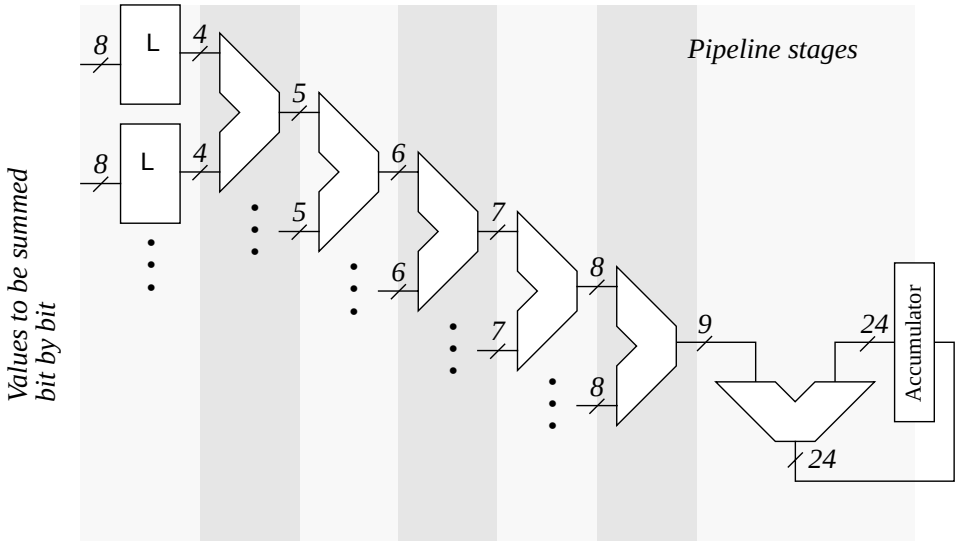

Figure 3 Part of a partly parallel adder tree to sum $M = 256$ numbers with maximally $q = 16$ bits each. "L" is the logic to add 8 bits into a 4-bit sum. Not shown are the pipeline registers between each layer of logic or adders.

- The cheapest solution, with respect to hardware, is to use a *bit-serial* adder tree completing the sum in $q + 2\log M$ steps. This solution utilizes $M - 1$ full adders with carry-save technique, and $2(M-1)$ latches. This method does not restrict the precision of the numbers to a specific maximum, and is faster for shorter numbers (i.e., smaller q). This structure can be extended with logic operations like global-and, global-or, and global-exclusive-or, but faster ways of doing global-or exist. It seems difficult, however, to directly extend a bit-serial adder tree to include reduction with minimum (or maximum). These operations can, however, be easily implemented by other means in a bit-serial fashion, cf. next section. If there are long wires between nodes or if they go outside a single chip, it is possible to reuse the wires in the adder tree for broadcast. This is accomplished by introducing backward bypasses on each full adder (FA), like the ones suggested by Pachanek et al. in [31].

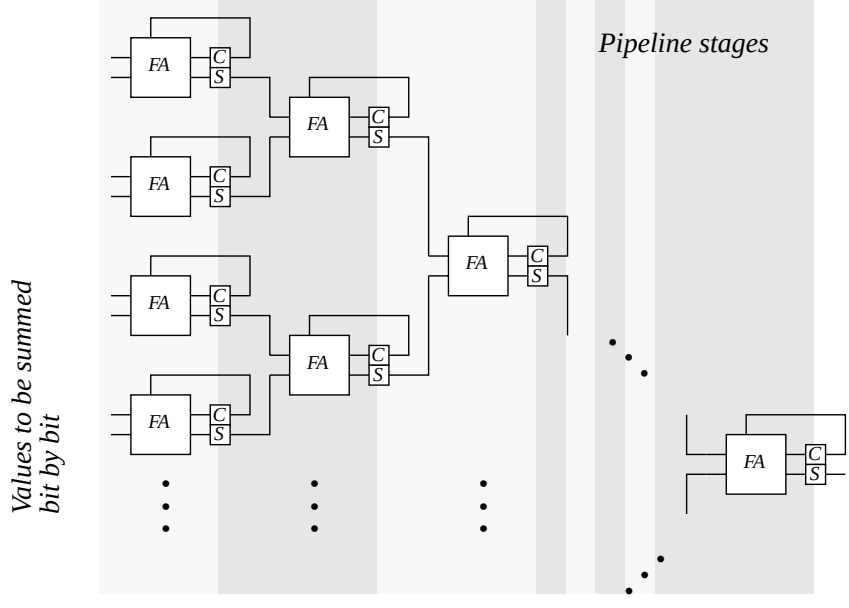

Figure 4 Part of a bit-serial adder tree to sum $M = 256$ numbers with maximally $q = 16$ bits each.

In Table 4 we have analyzed the three different variants of adder trees. The analysis has been done for a global-sum of $M = 256$ numbers with $q = 16$ bits each. This $M = 256$ was chosen as a maximum number of PEs on one chip. From the approximate analysis in this table we see that the partly parallel solution is the slowest method and thus not very interesting for further studies. We also note that the fully parallel and the bit-serial solution achieve similar speed. This is under the assumption that the bit-serial adder tree pipeline can be clocked four times faster than the fully parallel adder tree pipeline (which should be possible due to the simplicity of the bit-serial pipeline stages). The number of gates is not a very accurate measure of the area needed for each of the solutions, still it is an indication that the fully parallel solution is around 10 times larger than the partly parallel and bit-serial solutions. We

conclude that the bit-serial adder tree is the most promising way to expand an architecture with an adder tree. This is true even if the rest of the PEs run in bit-parallel mode, although some fast shift registers will then be needed for the conversion of parallel to serial data.

	Fully parallel	Partly parallel	Bit-serial
Steps to complete a global-sum	8	23	32
Most time consuming operation	add 23+23	add 9+24	FA
Unit delays Δ, for this operation	12	12	3
Speed (steps*Δ)	96	276	96
Number of parallel adders	255 (*16–23 bits in size*)	31 (*4–8 bits in size*)	—
Other logic	—	32 (*8 bits to 4-bit sum*)	255 FA
Number of gates	28930	4084	3060
Latches	4582	330	510

Table 4 Approximate analysis of the three adder tree alternatives. It is assumed that $M = 256$ numbers, having $q = 16$ bits each, are to be added. We denote the logic gate propagation delay with Δ. All methods are pipelined and for parallel adders four-bit carry-lookahead adders are used as described in [43].

Besides using the adder tree to calculate the output sum it can be used to help the LLS calculations if a normalized version of Eq. (1) is used and while using Hebbian or occurrence update of $\boldsymbol{w}_i$ a normalization is sometimes required. Additionally can the adder tree be useful if global rescaling is used (diagonal or scalar). Furthermore, an adder tree is not only advantageous for LLSs, it has also been found useful [42] for multilayer perceptrons with back propagation learning, on a SIMD processor array.

Using the local activity

An interesting prospect is to see if the local activity present in LLSs can be exploited to speed up the reduction operations. If we assume that a reasonable maximum is 2048 nodes (otherwise it could be split into multiple networks), then a bit-serial global-sum of 16-bit numbers takes $16 + 11 + 11 = 38$ steps to calculate. If we assume that only 1-2% of the nodes are active, the sum would minimally take $16 + 6 + 6 = 28$ steps. That is, the potential gain is relatively small. This is true at least if we keep the mixed parallel implementation and use an adder tree. However, in [28] it is shown that there are certain situations where the need for an adder tree can be avoided. This is accomplished by using a *transposed mapping*, where a node parallel mapping is used throughout the whole implementation as shown in Figure 5b.

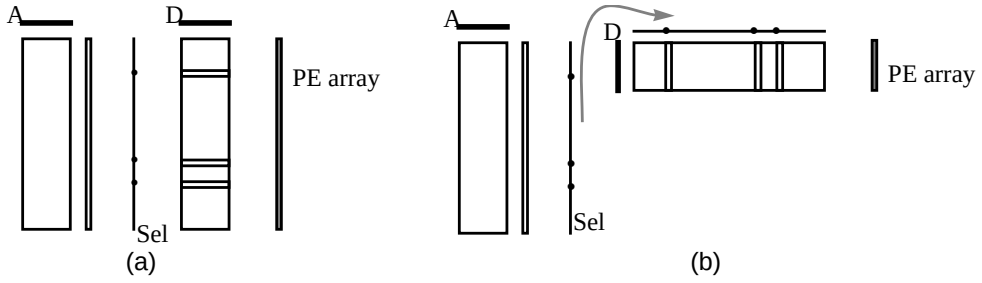

Figure 5 (a) shows a mixed parallel solution, used in this paper, while (b) shows the transposed mapping (node parallel) useful for the SDM algorithm, cf Figure 1b. The adder tree needed in (a) is not shown.

This was developed for the sparse distributed memory (SDM) model where no adaptation of the kernel positions takes place. SDM is designed to have very few active nodes $|A|$ and often works as an associative memory with many outputs. These are also the reasons why the mixed mapping works well for SDM. It becomes possible to iterate over each active node, and the PEs will only operate on local data. Given $|A| < m\log M$ this transposed mapping will compute the output sum faster than a mixed parallel mapping will. If for example $|A| < 0.1M$, that is, we only have 10% active nodes, then given $M = 128$ the transposed mapping will be faster if $m > 2$, and given $M = 512$ it will be faster if $m > 6$.

This solution needs m additional PEs. Depending on the memory each PE needs, it might be possible to reuse the "first layer PEs" for this new task. For each new PE we need M extra storage locations. There are also other things to consider, for example, means to translate the active PEs to addresses for the "second layer PEs" is needed. If the "first layer PEs" are reused for problems with few outputs then only a small number of assigned PEs will take part in the computations, leading to low efficiency.

We can conclude that for cases where the transposed mapping is not efficient we still want to have an adder tree available for the global-sum operation. Anyhow, means to implement transposed mapping should be provided as there is a lot to be gained from that mapping in certain cases.

4.3 Finding a minimum

Finding a minimum or reduction with a minimum operator is similar in structure to the other reduction operations like global-sum or global-or. The main difference is that we are not only interested in the resulting minimum value, but also in the index of the PE containing that value. As mentioned before, Raynolds et al. have described a general reduction network [39] capable of all important reduction operations including minimum. For the minimum operation they have added an extra path in the tree structure to be able to send along the indexes of the values reduced. The comparison is accomplished using a subtraction and the sign decides which value and tag to send forward. But, in correspondence to the global-sum, there are also bit-serial ways of finding minimum. Inspired by the bit-serial sorter by Afghahi [1] we now suggest a bit-serial minimum (or maximum) circuit.

The basic element is the bit-serial minimum or maximum filter found in Figure 6a. The input A and B are transmitted serially, with the most significant bit (MSB) first. Figure 6b shows the state diagram of the element. The reset signal sets the state back to the $A = B$ state. While there is no difference on the inputs there is no change in state. However, if a difference is detected the element either goes to the $A > B$ or the $A < B$ state as indicated in Figure 6b. In both these states the output transmitted will also depend on whether the minimum or maximum is wanted.

These basic elements are then connected into a binary tree structure.

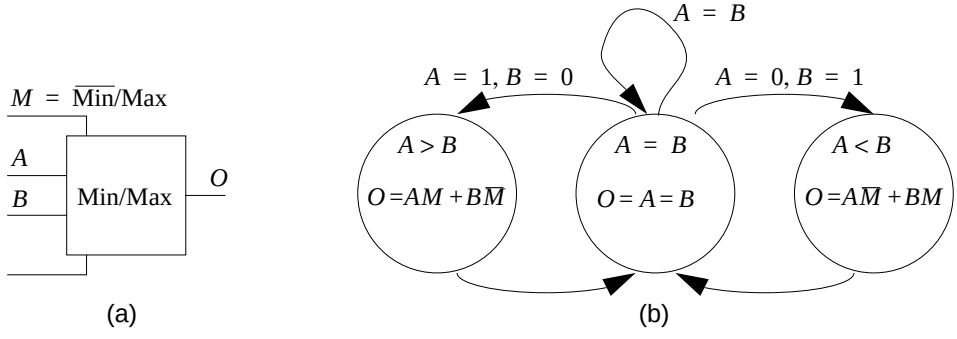

Figure 6 (a) A bit-serial min/max element of a binary tree. (b) State diagram of the min/max element.

As mentioned above the desired result of this operation usually is the index of the winning node instead of the minimum value. This can easily be achieved just by transmitting the index after the values to be compared. A nice by-product is that an automatic resolution of ties is also attained. The total number of steps will then be $q + 2\log M$ (the index has a length of $\log M$ bits). For the whole adder tree $M - 1$ of these elements are needed, where each element is not much more complex than a full adder.

Finding the runner-up, which is needed for some of the competitive learning methods, is easy as well. We just have to use a sorter element [1] as the root of the tree. The sorter has the same states as the min/max element but has two outputs: high and low.

Another way to find the maximum (which easily can be converted to finding the minimum) is described in [30] where a global-or network is used. The search starts by comparing the most significant bit for all values. It determines if any PE has a one using a global-or operation, in that case all PEs with zeros are turned inactive. For the case that all PEs show a zero, the search just continues to the next bit without changing the activity of any PE. The search continues in the next bit position and so on, until all bit positions have been treated. The time for this search is, in principle, independent of the number of data compared; it depends only on the data length q. The computation of the global-or will however depend on the number of data, and there is also a need for a select-first network [12] to resolve multiple maxima which will also depend on the number of data used. The total number of steps for the worst case is then $q\,(T_{GO} + 1) + T_{SF} + \log M$, where T_{GO} is the time to do global-or, T_{SF} is the time to do select-first, and $\log M$ is the index length in bits.

Comparing the two bit-serial methods to search for a minimum we find that the first bit-serial method is to be preferred if $qT_{GO} + T_{SF} > \log M$. As this is true under most circumstances (note that T_{GO} is sometimes even equal to $\log M$) the first method is promoted. This method also seems to put less demand on the controller.

5. Architecture requirements (summary)

In this Section we summarize the arithmetic and memory demands LLSs have on the processing elements (PEs). We also summarize our findings regarding communication networks, and also discuss what type of control is needed. An overview of the connection between different parts of the architecture is found in Figure 7.

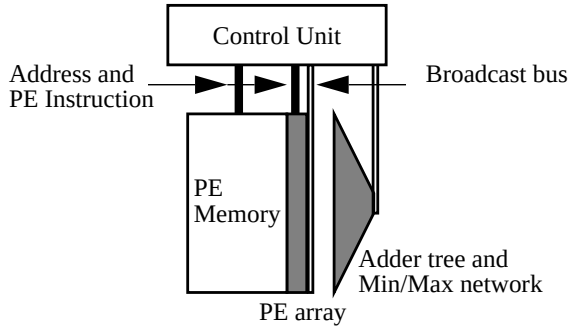

Figure 7 A simplified view of the suggested architecture. This is an array of processing elements (PEs) controlled by a single controller (SIMD). Besides a broadcast bus an adder tree and a minimum/maximum network have been added. Inter chip or inter module communication structures are not shown.

Note that the fact that the architecture is a *linear* array is not really used while implementing the LLS model. It could instead be considered as a set of PEs. The locality concept in the input space is usually not reflected in the order of the nodes.

The number of PEs should maximally be around 2048. If more nodes are desired by the LLS algorithm the nodes should be split into separate modules instead. If modern VLSI technology is used 128 – 256 PEs could be fitted onto one chip, possibly even with enough memory on-chip for some low level virtualization. This large number of PEs is easier to accomplish if low complexity PEs with bit-serial arithmetic are used. Fewer PEs or an off-chip memory solution might be required if bit-parallel PEs are to be used.

Arithmetic capability for PEs

Support for multiplication and addition is essential for ANN computations in general and that is true for LLSs as well. A useful addition would be support for absolute values. (Easily implemented but depends on the number representation used.) If low precision is to be used it also seems useful to have a soft overflow logic, that is, overflow should remain at max and not “swing around”.

A general way to support the implementation of radial functions is to use a table lookup. This solution will however require a more complex PE, and more memory per PE. This drawback and the fact that we can usually make a good approximation using only a few (say 2-10) multiplications and additions lets us conclude that support for table look-up is not essential.

The threshold logic unit operation needs a comparison which can be implemented using a subtraction and an “if then else” control.

Memory capacity for PEs

If we use a hyperbolic RBF (i.e. diag $[s_k]_i$) we will need memory for $3d + 2m + 9$ variables per PE. And for RCPL with counterpropagation weight adaptation we need memory for $2d + 2m + 7$ variables per PE. If a momentum term is used, then the memory needed is almost doubled. Still the memory needed is not very high per PE. This means that we can assume that the memory requirements will not be the bottleneck if we want to use the concept of virtual PEs as discussed in the end of Section 4. This assumption is true as long as we use external memory and do not use a very high level of virtualization (say above 32).

Communication

Besides broadcast (which is available on any SIMD computer) there is a need for an adder tree to support global summation, and a minimum/maximum network to support winner-take-all algorithms. These two reduction networks seem to be most efficiently implemented using a bit-serial approach. If the processors are distributed to multiple chips the communication between the chips needs to be considered. Thus, if there is a large speed difference between on-chip and off-chip communication/computation buffers will be required together with additional support from the controller. The more specialized approach with the transposed mapping also needs some support, for instance using a select-first structure.

If general communication networks like hypercube networks or mesh networks are available an efficient way to calculate the reduction operations discussed in this paper can be found. But for LLSs the flexibility of these networks is not needed and the extra hardware required would be unused most of the time.

Control

We have assumed that a SIMD computer, that is, a single controller for all PEs, should be sufficient for LLSs and found nothing to contradict this assumption. Besides the obvious operation to send out instructions and addresses, the controller, assuming a mixed parallel implementation, also has to:

- support a broadcast of the input vector (d steps),
- be able to receive the output sum from the adder tree (m times), calculate the difference to the target vector (m steps), and broadcast the difference to the PEs (m times); this step can also be done as a broadcast from the adder tree to all nodes (m times) and a broadcast of the target vector from the controller (m times).

As mentioned in the end of Section 4 the use of virtual PEs is a desirable method to reduce the number of PEs needed and support for virtual PEs increases the flexibility of the architecture. However, support for virtual PEs also demands more of the controller, for example, implementation of the reduction operation over all virtual PEs.

Fine grain PEs using bit-serial arithmetics allow for a shorter clock cycle; unfortunately this also puts higher demands on the controller. At a certain speed the controller might need to be split into one on-chip “loop” controller section and one off-chip main (slower) controller section, that is, a hierarchical control.

As addressing autonomy is not needed (unless table lookup is to be used for kernel function calculation) the control unit will generate the PE memory addresses.

5.1 REMAP

REMAP ("Real-time, Embedded, Modular, Adaptive, Parallel processor project") [7] is a joint project between Luleå University of Technology, Chalmers University of Technology, and Halmstad University (all in Sweden). The first prototype has been implements using programmable logic (FPGA) to facilitate architectural experiments [24]. This prototype mainly uses bit-serial PEs which are organized as a linear processor array controlled in SIMD fashion. The implemented communication structures are nearest neighbor and broadcast. The PEs are ordinary bit-serial PEs capable of doing addition subtraction, and, or, not, exclusive-or, load register and a tagged store. In earlier studies we have found that in order to support ANN computations the basic PEs need to be extended with a bit-serial multiplier [12] (for MLP [42] and SOM [27]) or a bit-counter (for SDM [28]).

To support LLSs this basic REMAP architecture only needs to be expanded with support for reduction operations (adder-tree and min/max network). This support for reduction operations also can be useful for multi-layer Perceptrons trained with back-propagation. In an earlier paper [30] we suggested that temporary sums could be circulated among the PEs to achieve the same result. But, for LLSs with many more nodes in the hidden layer compared to the output layer, that solution is not as efficient as the solution suggested in this paper.

6. Related work

In the following sub-sections we will discuss some of the more promising hardware implementations which can support LLS computations. Further discussions on parallel hardware for ANN in general, and some special LLS variations like sparse distributed memory (SDM) and self-organizing maps (SOM) can be found in our papers [26, 27, 30].

6.1 INTEL and Nestor

Intel and Nestor have developed the Ni1000 chip [16, 20], implementing variants of restricted Coulomb energy (RCE) [6] and probabilistic neural network (PNN) [41] algorithms. It has two main parts, where the first part consists of 512 distance calculating units, and the second is a DSP-like mathematical unit. The first part, as the name indicates, calculates a distance between an input vector and kernel centers. They can only calculate a city-block distance (L_1). These units are organized in a node parallel fashion and operate at 40 MHz. Up to 1024 prototypes can be stored on the chip, each prototype with a maxima length of 256 values with 5-bit precision. The mathematical unit is used to calculate the kernel functions (Gaussian or TLU) and the output sum. It uses 16 bit floating-point numbers for its calculations. It can operate at 40000 patterns/s in a feedforward mode.

This chip only implements a subset of the LLS variations (e.g. only L1 distance and only one form of receptive fields). Still, for problem areas where these restrictions can be accepted the Ni1000 chip performs well.

6.2 ZISC

ZISC036 is the first in a family of ANN chips developed by IBM [11, 19]. It implements an LLS variation similar to an RCE network. This chip implements 36 neurons organized in a node parallel fashion and operating at 20 MHz. The chips can easily be cascaded to larger configurations. Each node contains a prototype with a maximal length of 64 values using 8-bit precision. Only L_1 or L_∞ distance measurements can be used and the kernel function is implemented as a threshold. Means to extract the k closest nodes are also available. Learning is done in an RCE fashion where an unrecognized input is attached to an unassigned node and only the receptive field sizes can be adapted (actually they can only be reduced). In ZISC036 there is no support to implement virtual PEs.

While ZISC036 is being implemented in conservative VLSI technology (1 µm) it does not seem to improve much on the Ni1000 chip besides being designed to be easily extended to large multi chip devices.

6.3 CNAPS

The CNAPS chip from Adaptive Solutions Inc. is a chip developed for ANN calculations [13, 14] and has an architecture similar to the one suggested in this paper. It has been used for many variants of ANN with impressing figures {9.6 GCPS and 2.3 GCUPS for multilay-

er perceptrons (MLP) with back propagation (BP) learning, with 8 chips [25]}. It contains 64 DSP-like PEs forming a linear SIMD array, where each PE is capable of doing multiply-and-accumulate at a rate of 25 MHz. Memory for the weights is found on the chip (4Kb/PE).

There is a global maximum selection available which is useful for the SOM implementation. There is, however, no support for a global-sum. This probably means that many LLS variations (with more than one active node) will be difficult to implement efficiently. The only LLS variation found for CNAPS is an implementation of SOM [15]. This implementation shows relatively poor performance [27]. Even if Pulkki [35] has almost tripled these performance figures, we do suspect that CNAPS needs to be complemented with an adder tree to fully support the LLS model.

6.4 Other digital solutions

There is a number of other general ANN computers that could also be efficient for LLS algorithms. The most promising ones seem to be the Siemens SYNAPS (MA-16)[36, 37, 38] and ICSI's SPERT [5] and their upcoming CNS [4]. As both ICSI's and Siemens' designs use off-chip memories, larger problems can be simulated on them compared to what is possible on Ni1000, ZISC, and CNAPS. The design of SYNAPS and CNS are directed towards batch oriented training which suggests that the Ni1000, ZISC, and CNAPS will be better for on-line (embedded) computing.

Moreover, we note that there are general purpose computers, especially the machines that have support for reduction operations, like CM-5 and Intel Paragon, which potentially can achieve good performance for LLS algorithms. However, the communication latencies in these computers indicate that they are better in off-line situations, where batch oriented training can be used.

For binary versions of the RCE or the SDM algorithms the SPISM (standard pinout implementation of smart memory) chip could be interesting [40]. But the restricted ways to form outputs severely reduce its general usefulness.

6.5 Analog hardware.

Analog VLSI for LLS computations seems promising for the future. The high speed of analog VLSI is attractive for many ANN calculations. This is also true for LLSs. Especially the global-sum seems to benefit much from an analog implementation. But currently analog technology has too little flexibility for most real-world usage. Better means to train on-line, permits different LLS variations, allow higher precision when desired, and permit multi-chip extensions are some of the things needed before analog technology can be put into practical use. The most promising application area for analog VLSI seems to be close to the sensors where the input can be kept analog all the way through.

Despite the problems there are some research chips manufactured to do RBF calculations. Some have concentrated on the distance calculation and the kernel function. One such study is the analog computation of the Euclidean norm with a varying width Gaussian made by Churcher et al. [9]. Others, like Watkins et al., have implemented a hybrid computer for

RBF [44, 45]. They use analog technology for the first steps: calculating a Euclidean distance and a Gaussian kernel with variable width. For the (weighted) summation they use an ordinary DSP (M56000), which is also used for learning. The computation of the Gaussian was later moved to the DSP due to the large range of receptive field sizes the algorithm needed. A completely analog RBF VLSI chip is described by Anderson et al. in [3].

7. Conclusion

In this paper we have analyzed the implementation aspects of localized learning systems (LLSs) presented in [29]. The objective has been to suggest suitable parallel computer architectures for the LLSs.

Our hypothesis that a SIMD (single instruction stream multiple data stream) computer array should be suitable for LLSs has been reinforced in this study. As we think that an ANN computer should support many ANN models and variations, the architecture must be programmable, either in hardware or in software. Among the reported implementations of LLSs only our own REMAP and CNAPS combine the required programmability with high performance on-line learning capability. Besides the obvious support for addition and multiplication the radial function calculation will need some consideration. A mixed parallel implementation has been found to best suit the LLS algorithms as long as we demand on-line operation. Besides broadcast communication the LLS algorithm will need support to do global-sum and find global maximum. Three implementations of global-sum are identified and studied. It is established that this operation can be performed bit-serially at the same speed as bit-parallel solutions with a fraction of the hardware cost. For the global-minimum operation a new bit-serial structure is proposed. This new min/max network has the advantage of not needing a global-or network as the standard bit-serial way of finding minimum does. This also results in a speed advantage in most cases.

Even for very large LLSs (d = 64, m = 64, and M = 2048) the suggested architecture can accomplish update rates at many thousand updates per second, and that on a processor array with only 256 PEs operating at 25 MHz.

We had hoped to make better use of the locality aspect of LLSs than we finally could achieve. But on the other hand we can note that the suggested architecture, especially the addition of an adder tree, makes the implementation of multilayer perceptrons with back propagation learning efficient. Only in some more specialized cases we can, via a transposed mapping, use the locality to achieve better performance than by using an adder tree.

7.1 Acknowledgments

The author would like to thank Prof. Bertil Svensson Chalmers University of Technology, Assoc. Prof. Lennart Gustafsson Luleå University of Technology, Assoc. Prof. Glenn Jennings Luleå University of Technology, and Lic. Tech. Per Ödling Luleå University of Technology for valuable discussions.

8. References

[1] Afghahi, M., "A 512 16-b bit-serial sorter chip," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 10, pp. 1452-1457, 1991.

[2] Albus, J. S., *Brains, Behavior, and Robotics*, Petersborough, NH, USA: BYTE/McGraw-Hill, 1981.

[3] Anderson, J., J. C. Platt and D. B. Kirk, "An analog VLSI chip for radial basis functions," in *Neural Information Processing Systems 5*, C. L. Giles, S. J. Hanson and J. D. Cowad Eds. Denver, CO, USA, 1992, pp. 765-772.

[4] Asanovic, K., et al., "CNS-1 architecture specification," Tech. Rep. TR-93-021, International Computer Science Institute, 1994.

[5] Asanovic, K., J. Beck, B. E. D. Kingsbury, P. Kohn, N. Morgan and J. Wawrzynek, "SPERT: a VLIW/SIMD neuro-microcomputer," in *International Joint Conference on Neural Networks*, Baltimore, 1992, vol. 2, pp. 577-582.

[6] Bachmann, C. M., L. Cooper, A. Dembo and O. Zeitouni, "A relaxation method for memory with high storage density," *Proceedings of the National Academy of Sciences*, vol. 84, pp. 7529-7531, 1987.

[7] Bengtsson, L., A. Linde, B. Svensson, M. Taveniku and A. Åhlander, "The REMAP massively parallel computer platform for neural computations," in *Third International Conference on Microelectronics for Neural Networks (MicroNeuro '93)*, Edinburgh, Scotland, UK, 1993,

[8] Blank, T. and J. R. Nickolls, "A grimm collection of MIMD fairy tails," in *Frontiers of Massively Parallel Computation (Frontiers ´92)*, H. J. Siegel Ed., McLean, Virginia, USA, 1992, pp. 448-457.

[9] Churcher, S., A. F. Murray and H. M. Reeckie, "Programmable analogue VLSI for radial basis function networks," *Electronics Letters*, vol. 29, no. 18, pp. 1603-1605, 1993.

[10] Cormen, T. H., C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[11] Eide, Å., T. Lindblad, C. S. Lindsey, M. Minerskjöld, G. Sekhniaidze and G. Zsékely, "An implementation of the zero instruction set computer (ZISC036) on a PC/ISA-bus card," in *WNN/FNN workshop*, Washington DC, 1994,

[12] Fernström, C., I. Kruzela and B. Svensson, *LUCAS Associative Array Processor - Design, Programming and Application Studies*, vol. 216 of *Lecture Notes in Computer Science*, Berlin: Springer Verlag, 1986.

[13] Hammerstrom, D., "A VLSI architecture for high-performance, low-cost, on-chip learning," in *International joint conference on neural networks*, San Diego, 1990, vol. 2, pp. 537-543.

[14] Hammerstrom, D. and E. Means, "System design for a second generation neurocomputer," in *International Joint Conference on Neural Networks*, Washington D.C., 1990, vol. 2, pp. 80-83.

[15] Hammerstrom, D. and N. Nguyen, "An implementation of Kohonen's self-organizing map on the Adaptive Solutions neurocomputer," in *International Conference on Artificial Neural Networks*, T. Kohonen, et al. Eds. Helsinki, Finland, 1991, vol. 1, pp. 715-720.

[16] Holler, M., et al., "A high performance adaptive classifier using radial basis functions," in *Government Microcircuit Applications Conference*, Las Vegas, Nevada, November 9-12, 1992,

[17] Hudak, M. J., "RCE classifiers: theory and practice," *Cybernetics and Systems*, vol. 23, pp. 483-515, 1992.

[18] Hwang, K., *Computer Arithmetic: Principle, Architecture, and Design*, New York: John Wiley & Sons, 1979.

[19] IBM Microelectronics, “ZISC036 user's manual (preliminary),” IBM France, Component Development Laboratory, 1994.

[20] Intel Corporation, “Ni1000 Specifications,” Intel Corp. and Nestor Inc., 1993.

[21] Kanerva, P., *Sparse Distributed Memory,* Cambridge, MA: MIT press, 1988.

[22] Kohonen, T., “The self-organizing map,” *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464-1480, 1990.

[23] Kung, S. Y., *Digital Neural Networks*, Englewood Cliffs, N.J.: Prentice-Hall, 1993.

[24] Linde, A., T. Nordström and M. Taveniku, “Using FPGAs to implement a reconfigurable highly parallel computer,” *Field-Programmable Gate Array: Architectures and Tools for Rapid Prototyping; Selected papers from: Second International Workshop on Field-Programmable Logic and Applications (FPL'92), Vienna, Austria*, H. Grünbacher and R. W. Hartenstein Eds. New York: Springer-Verlag, pp. 199-210, 1992.

[25] McCartor, H., “Back propagation implementation on the Adaptive Solutions CNAPS neurocomputer chip,” in *Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S. Touretzky Eds. Denver, CO, USA, 1990, pp. 1028-1031.

[26] Nordström, T., “Sparse distributed memory simulation on REMAP3,” Res. Rep. TULEA 1991:16, Luleå University of Technology, Sweden, 1991.

[27] Nordström, T., “Designing parallel computers for self organizing maps,” in *DSA-92, Fourth Swedish Workshop on Computer System Architecture*, Linköping, Sweden, 1992,

[28] Nordström, T., “Hardware for sparse distributed memory simulations,” *to be submitted*, 1995.

[29] Nordström, T., “On-line spatially localized learning systems, part I - model description,” *to be submitted (Also available as Res. Rep. TULEA 1995:1, Luleå University of Technology, Sweden)*, 1995.

[30] Nordström, T. and B. Svensson, “Using and designing massively parallel computers for artificial neural networks,” *Journal of Parallel and Distributed Computing*, vol. 14, no. 3, pp. 260-285, 1992.

[31] Pachanek, G. G., S. Vassiliadis and J. G. Delgado-Frias, “Digital neural emulators using tree accumulation and communication structures,” *IEEE Transaction on Neural Networks*, vol. 3, no. 6, pp. 934-950, 1992.

[32] Platt, J. C., “Learning by combining memorization and gradient descent,” in *Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S. Touretzky Eds. Denver, CO, USA, 1990, pp. 714-720.

[33] Poggio, T. and F. Girosi, “A theory of networks for approximation and learning,” A.I. Memo 1140 (first released 1991), Massachusetts Institute of Technology, 1994.

[34] Powell, M. J. D., “Radial basis functions for multivariable interpolation: a review,” in *IMA Conference on Algorithms for the Approximation of Functions and Data*, RMCS, Shrivenham, UK, 1985, pp. 143-167.

[35] Pulkki, V., Helsinki University of Technology, 1994, Personal communication.

[36] Ramacher, U., “SYNAPSE — a neural computer that synthesizes neural algorithms on a parallel systolic engine,” *Journal of Parallel and Distributed Computing*, vol. 14, no. 3, pp. 306-318, 1992.

[37] Ramacher, U., et al., “Design of a 1st generation neurocomputer,” *VLSI Design of Neural Networks*, U. Ramacher and U. Rückert Eds. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1991.

[38] Ramacher, U., et al., “Multiprocessor and memory architecture of the neurocomputer SYNAPSE-1,” in *International Conference on Artificial Neural Networks-93*, Amsterdam, The Netherlands, 1993, pp. 1034-1039.

[39] Raynolds, P. F. J., C. M. Pancerella and S. Srinivasan, "Design and performance analysis of hardware support for parallel simulations," Research Report CS-92-20, University of Virginia, 1992.

[40] Smith, D., M. Shetti, M. Harward, W. Bean, R. Pawate and G. Doddington, "A VLSI implementation of the Nestor RCE neural network," *Texas Instruments Technical Journal*, vol. 7, no. 6, pp. 34-41, 1990.

[41] Specht, D. F., "Probabilistic Neural Networks," *Neural Networks*, vol. 3, no. 1, pp. 109-118, 1990.

[42] Svensson, B. and T. Nordström, "Execution of neural network algorithms on an array of bit-serial processors," in *10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*, Atlantic City, NJ, USA, 1990, vol. II, pp. 501-505.

[43] Waser, S. and M. J. Flynn, *Introduction to arithmetic for digital systems designers*, Holt, Rinehart and Winston, 1982.

[44] Watkins, S. S., P. M. Chau and R. Tawel, "A radial basis function neurocomputer implemented with analog VLSI circuits," in *IJCNN International Joint Conference on Neural Networks*, Baltimore, MD, USA, 1992, vol. 2, pp. 607-612.

[45] Watkins, S. S., P. M. Chau, R. Tawel, B. Lambrigtsen and M. Plutowski, "A hybrid radial basis function neurocomputer and its applications," in *Neural Information Processing Systems 6*, Denver, CO, 1993,

[46] Wong, Y., "How Gaussian radial basis functions work," in *International Joint Conference on Neural Networks*, Seattle, WA, USA, 1991, pp. 133-138.

[47] Xu, L., A. Krzyzak and E. Oja, "Rival penalized competitive learning for clustering analysis, RBF net, and curve detection," *IEEE Transactions on Neural Networks*, vol. 4, no. 4, pp. 636-649, 1993.

Nordström, T., “Hardware for sparse distributed memory simulations,” to be submitted for publication, 1995. An earlier version is available as a research report: Nordström, T., “Sparse distributed memory simulation on REMAP3,” Res. Rep. TULEA 1991:16, Luleå University of Technology, Sweden, 1991.

Hardware for Sparse Distributed Memory

Tomas Nordström

Division of Computer Science & Engineering
Luleå University of Technology, Sweden
E-mail: tono@sm.luth.se

ABSTRACT

Sparse distributed memory (SDM) has been used to solve problems in areas such as speech recognition, pattern matching and temporal encoding. It has been found to be suitable for implementation on parallel computers. SDM can be described in terms of an artificial neural network model as well as a very large computer memory.

The computer model we map SDM onto is among the simplest of them all: a bit-serial linear array with SIMD control. A prototype, called REMAP, is used as an example of this kind of architecture.

The implementation of a normal problem using 256 REMAP processing elements is estimated to run 10 times faster than the normal Connection Machine (CM-2) simulation, where 8k processing elements are used. This is due to an efficient mapping of the SDM model onto our computer, and to the possibility of configuring the architecture especially for the type of calculations needed for SDM.

1. INTRODUCTION

In problem areas like vision, optimization and speech processing, humans often perform well. Still, these problems have been very hard for ordinary computers. Inspired by the massively parallel and highly interconnected structure of the brain, artificial neural networks (ANN) have been suggested to solve these problems.

Localized learning system (LLS) models [22] is a large group of ANN models which have attracted interest lately. All the LLS models are feed-forward networks using an expanded representation (large number of hidden nodes) [36], and having the important feature of localized activity and learning. These properties have been shown to make efficient parallel computer implementations possible [23]. Examples of LLS models are: generalized radial basis functions (GRBF) [25, 26], self-organizing feature maps (SOFM) and learning vector quantization (LVQ) [18], probabilistic neural network (PNN) [35], and cerebellar model arithmetic computer (CMAC) [1]. This paper will concentrate on another LLS model, the sparse distributed memory (SDM) developed by Kanerva [13]. Due to the binary nature of SDM some special care is needed for efficient implementations.

SDM has been used in pattern matching and temporal sequence encoding [12, 13, 15]. It can also be used as an associative memory [14, 40]. Kanerva [13, 14] has argued that SDM is a biologically plausible model. Prager, Fallside and Clarke have used a modified version of SDM for real-time speech recognition [5, 27, 28]. Rogers [30] stresses that sparse distributed memory is an ideal artificial neural network for massively parallel computer implementation.

In this paper we try to identify architectural properties which are important for the simulation of SDM. We also estimate the performance on a bit-serial linear processor array called REMAP. Further, we review other SDM implementations on special hardware and on the Connection Machine. Finally, we draw some conclusions about the suitability of REMAP for the simulation of SDM.

2. THE REMAP COMPUTER

REMAP (Real-time, Embedded, Modular, Adaptive, Parallel processor project) and is a long-term project addressing questions related to the usage of massively parallel, distributed computing in embedded systems [3, 7, 19, 38]. Of specific interest is the potential of making "action-oriented" systems [2] that interact with the environment by means of highly parallel sensors and actuators. The project is done as a joint project between Luleå University of Technology, Chalmers University of Technology, and Halmstad University.

Within the current project, a small prototype of a software configurable processor array module has been implemented. Different variations are possible by reprogramming. However, this possibility has not been fully used due to the poor quality of the design tools. Still, an architecture tuned for neural network computations, including a fast bit-serial multiplier, has been designed [19]. Between 8 and 16 processing elements (PEs) can be configured in one logic chip Xilinx 4005. The PEs typically have four one-bit registers for ordinary bit-serial computations and are not very different from the PEs of array computers like DAP, Blitzen or Connection Machine.

In this paper two models of the REMAP PEs are analyzed. One uses the basic REMAP PE without extra supporting hardware. This PE includes a bit-serial multiplier needed for many other ANN algorithms. However, for SDM no such multiplier is needed and for the second model, the multiplier is replaced by a counter preceded by an exclusive-or gate. This speeds up the selection phase by three to four times.

3. THE SPARSE DISTRIBUTED MEMORY MODEL

The Sparse Distributed Memory (SDM) model developed by Kanerva [13] can be described in two ways: as a computer memory or as a two layer feedforward network. By viewing SDM as a computer memory it can be compared to the random access memory (RAM) of conventional computers. Both SDM and RAM store data at an address accessed by a "reference address." The difference from conventional RAM is that instead of having 32-bit addresses, SDM may have 1000-bit addresses. As it is impossible to have a physical memory with 2^{1000} addresses, only a small part of it can be populated, i.e. it will be *sparsely* populated. Reference addresses with no physical addresses available must somehow be associated with one or many other (physical) addresses. Different solutions exist for different types of associative memories. SDM is a *distributed memory* model (as the name indicates), and each reference address is associated with many physical addresses. To be able to store more than a single data item in one location address, the data is stored in counters instead of one bit cells as in RAM, see Figure 1.

In the basic form the physical location addresses are evenly distributed as random points in the global address space. If the reference addresses are not evenly distributed i.e. the inputs are correlated, the location addresses should be distributed according to the probability distribution of the reference address [6, 17]. This can be achieved in many ways. Danforth [6] used a collection of selected templates, Rogers used genetic algorithms in [31] and Saarinen et al. [34] described a method to use a version of Kohonen's self-organizing feature maps to let the physical addresses first "self-organize" into the probability distribution, and after that use it in the usual way. Another way to deal with correlated data, suggested by Kanerva [16], is to weight each bit separately in the input address to improve the separability of patterns. All these variations fit nicely into the concept of localized learning system (LLS), and by studying LLS many other variations can be found [22, 23]. However, in this report only the original algorithm will be described, but we note that most of the variants can be implemented equally efficiently on the suggested hardware.

Instead of looking for an exact match for the address, SDM will look for location addresses that are close to the reference address. By close we mean the shortest Hamming distance (i.e. the minimum number of bits that differ). Usually more than one location is considered as close and therefore selected i.e. the memory is distributed. When reading from the memory all the selected counters are summed and thresholded ($\sum<0$ corresponds to 0, $\sum\geq0$ to 1). Storing is done by incrementing or decrementing the selected counters (0 corresponds to a decrement and 1 to increment). The procedure is summarized in Algorithm 1.

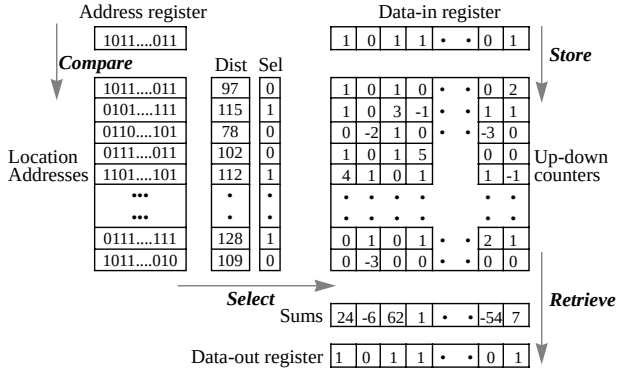

Figure 1 The organization of a Sparse Distributed Memory as an array of addressable locations. Note that the address as well as the data can be of lengths of hundreds of bits, still there are only a small number (usually less than a million) of memory locations.

Algorithm 1 The SDM algorithm

Training the network (i.e. Writing to the memory):

1. The address register is compared to the location addresses and the distances are calculated.
2. The distances are compared to a threshold and those below the threshold are selected.
3. In the selected rows,

   where the data-in register is 1 the counter is incremented,

   where the data-in register is 0 the counter is decremented.

Recall from the network (i.e. Reading from the memory):

1. The address register is compared to the location addresses and the distances are calculated
2. The distances are compared to a threshold and those below the threshold are selected.
3. The selected rows are added columnwise.

   Where the sum is greater than zero the data-out register is set to one,

   else it is set to zero.

The choice of radius (or threshold), r, and the address length in bits, α, together with the number of location addresses, N, will determine the performance of the memory [13]. Given $r = 111, \alpha = 256, N = 8192$ the mean number of active positions will be $\bar{A} = 160$, and if we instead have $r = 451, \alpha = 1000, N = 2^{20}$ it leads to $\bar{A} = 1124$.

4. MAPPING SDM ONTO AN ARCHITECTURE

Algorithm 1 can be mapped onto a computer architecture in many ways. There are two major ways to map SDM onto an array of PEs: rowwise and columnwise. It will later be found that, for some architectures, a mixture of the two is the best method of mapping.

4.1 The Rowwise Mapping

The rowwise mapping, in which each PE takes care of one physical location address and its data, is probably the most natural one. It is illustrated in Figure 2.

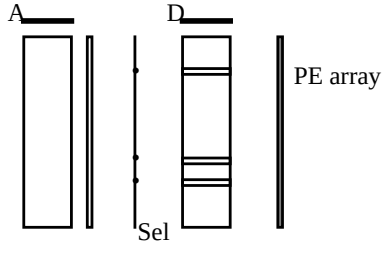

Figure 2 A stylized illustration of the rowwise mapping, compare to Figure 1.

Using the classification, introduced by Nordström and Svensson in [24], for implementation of ANN models, this mapping would be called node parallelism. Another type of parallelism called weight parallelism would correspond to having one PE for each address and data bit/counter. The large number of PEs and the massive communication needed makes weight parallelism unrealistic for large SDM models. (For example, a 1000-bit address and a 1000-bit data input field, with 10^6 location addresses, would need 2 10^9 PEs and a communication fan-in and fan-out of 10^6.)

The rowwise mapping is very efficient for the selection phase, that is, the first two steps in Algorithm 1. In this phase all the calculations needed can be done locally in each PE. If the number of PEs is the same as the number of location addresses the PE utilization can be 100%.

In the store/retrieve phase only a small portion of the processor array is actively taking part in the computation, because a relatively small number of PEs are selected in the selection phase (can be as low as 0.1-1%). This inefficiency can be removed if the counter array is transposed as described in the next two mappings.

During retrieve a summation of active PEs should be carried out across PEs (sum-reduction) and communication support for this type of operation must be available, for example, using an adder-tree.

4.2 The Columnwise Mapping

By “transposing” the address and counter array we get a columnwise mapping, see Figure 2.

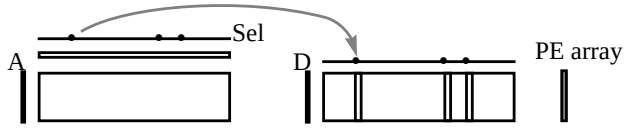

Figure 3 A stylized illustration of the columnwise mapping

Using columnwise mapping for the selection phase the counting of bits (Hamming distance calculation) must be carried out across PEs. This could be done by an adder-tree [8] but even then the rowwise mapping is found to be more efficient for this phase.

The store/retrieve phase (which needed sum-reduction across PEs in the rowwise mapping) can now be carried out locally, and no special communication support is needed. The PEs will also be utilized more efficiently. For each active position we can now update/use all PEs. Typically we have 256 data bits and only a mean of $\bar{A}$ =160 active positions. The net result for the store/retrieve phase is that we now do more work with fewer PEs!

When storing, support for doing subtraction and addition simultaneously would reduce the update time by half. This can easily be achieved in both bit-serial and bit-parallel architectures.

It should be noted that this mapping depends on the fact that the number of address and data register bits is often large. If this is not the case, as in the extreme case where only one bit is used for data, the rowwise mapping is more efficient.

4.3 The Transposed Model

By “transposing” only the counter array we get a mixed row- and columnwise model as in Figure 4. The transposed model combines the best parts of the two major mappings: rowwise mapping for the selection phase and columnwise for the store/retrieve phase.

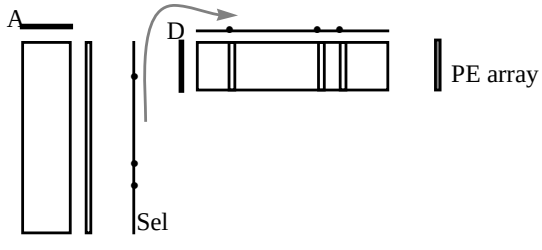

Figure 4 The “transposed” method of mapping the SDM algorithm into a linear array.

For this model it is natural to have as many PEs as there are data input bits. Typically this is between 256 and 1024 bits when using SDM as an associative memory. Thus a linear processor array of size 256 to 1024 PEs would be sufficient.

Unfortunately we introduce some new problems that have to be solved as well.

- The selection phase now has fewer PEs to utilize.
- We need to be able to "move" the active positions from rowwise to columnwise representation.

The Selection Phase Using a Transposed Model

Returning attention to the selection phase, knowing that the number of PEs for the store/retrieve phase is much lower than for the selection phase, we can solve the imbalance by:

- having special PEs for the selection phase, or
- using virtual PEs (thus multiplexing PEs), or
- increasing the number of PEs until a suitable trade-off is achieved.

Each method is discussed below.

Special PE chips.

To get maximal performance on the selection phase a special purpose PE array could be constructed in VLSI. The array would be specialized to perform Hamming distance calculations bit-serially. If the initial loading of location addresses is done serially there would be no need for each PE to have its own connection to the outside of the chip. After the Hamming distance calculation and the comparison to a radius is made an internal activation flag could be set. Ways to move this information to the controller are described in Section 4.3.1.

Mackie and Denker discuss a special purpose chip in [21] useful for the distance calculation suggested above. The number of PEs on a 0.9μm CMOS chip is 50, each storing 128 bits. A "best match list" generator is included, so after 2.5 μs a list of the 5 best matching units (out of 50) is produced.

Using special PE chips for the selection phase makes the total system heterogeneous. If we want a more homogenous computer array, virtual PEs or the addition of new arrays can be used instead.

Virtual PEs

By simply partitioning the location addresses into chunks of the array size we can achieve reasonable performance on not too large problems.

On a large problem, such as one with 10^3 input data bits and 10^6 location addresses, the selection phase will dominate the calculation time completely. Even if the PEs are fully utilized there are a lot of unused parallelism in the selection phase, and it becomes of interest to balance the phases.

Increasing the Number of PEs

Just adding more PEs for use in the selection phase would not gain anything for the store/retrieve phase, and utilization would go down. We could instead add a new array and have the

SDM computation partitioned according to Figure 5. This partitioning can be done repeatedly until the mean number of active location addresses gets too low (and the utilization rapidly goes down). The performance will increase with the number of modules added. The optimum number of PEs will depend on the performance needed for the problems being solved. The price one has to pay is to add a new controller for each new array. Each array should have as many PEs as there are bits in the data-in field.

This way of increasing the number of PEs leads to an architecture comprising multiple SIMD (single instruction stream multiple data stream) modules with the same or similar code controlling each array. The difference between the arrays is that different arrays will have different active location addresses and therefore different addresses for the counters. The number of active locations will also differ.

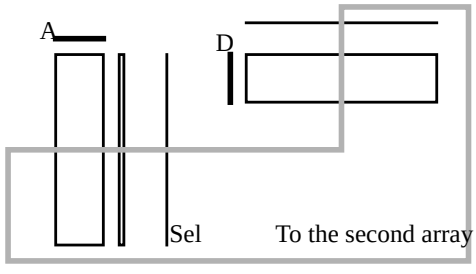

Figure 5 How to partition the transposed mapping into twice the number of PEs in a multi modular fashion.

4.3.1 Moving Data From Row to Column Representation

After the selection phase, using rowwise mapping, we have one bit in each location address indicating whether the corresponding data counter is to be active (selected) in the store/retrieve phase. The controller could find out which counters are selected in at least three ways:

1. One bit at a time could be shifted out of the array (using nearest neighbour communication). This will be done in the same number of clock cycles as there are location addresses.
2. If there exists a serial to parallel output like a corner-turner it could be used to reduce the time by a factor of 8 (for an 8 bit corner-turner). This is only useful if each PE is used for many location addresses, i.e. virtual PEs.
3. The last and quickest method is to use a select first network [8] to find the active ones and for each one send out its processor number to the control unit. This can be done in one or two cycles for each active location, plus the time to send the PE identification number.

Note that the order of the counters does not need to be the same as the order of the bits in the selection vector. The only requirement is that the same order is used from time to time and is the same for both store and retrieve.

5. THE TRANSPOSED MAPPING DONE BIT-SERIALLY

Many of the massively parallel processors like Blitzen [4], REMAP, and Connection Machine [10, 39] are bit-serial. That imposes some restrictions on what data types and calculations are suitable for them. By analyzing each step of the SDM algorithm we can determine the suitability of bit-serial arithmetic for this problem.

Hamming Distance Calculation.
The difference between the reference and the location addresses is obtained by an exclusive-or operation on each bit position. The distance is the number of ones in the difference. The sum can be calculated in α (number of bits in the address) steps if there is support for a counter in the architecture. If there is no counter the summation of bits can utilize the "recursive" structure of the many short sums in the beginning and few long sums in the end. The sum can be calculated in $\leq$1409 steps for 256 bits and $\leq$5723 steps for 1024 bits using a bit-serial adder. (These figures are for a summation starting with three bits at the time, and later partitioned into sums from 32 bits. Any controller overhead is not included.)

These types of calculations are very well suited for bit-serial computers compared to bit-parallel computers. Using a conventional microprocessor like MC68030 the Hamming distance of 256 addresses with 256 bits each could be computed in about 84000 clock cycles, using a table lookup for the bit counting. This should be compared to the number of steps above.

Compare to Threshold
In the store/retrieve phase of the algorithm the counters are compared with a threshold (the radius). This can be carried out bit-serially in $2b$ steps, where b is the length of the bit-counter (typically 16-20 steps).

Moving the Selection Vector to the Controller
All methods described in Section 4.3.1 could be used in a bit-serial architecture. Each method needs some "special" communication: nearest neighbor, corner-turner, or select/first. In almost all bit-serial architectures at least one is present. The number of steps will depend on what method is used to move the data and what kind of communication is available.

Add/Sub one to Selected Counters, Sum All Selected Counters
During the store phase the data-in register can be sent in parallel into the array. If no parallel I/O channels are available the data-in register can be shifted in bit by bit or by using a corner-turner. After the data is moved each processor will have one bit each of the data-in register. These data bits control the PE operation: add one or subtract one. This can easily and efficiently be accomplished, as implemented in Blitzen [4] just by adding an extra register bit controlling the addition-subtraction. Then one field of counters in the memory can be updated in just 2σ clock cycles (where σ is the counter length).

During the retrieve phase there is no need to input the data register. Instead the values of the active counters should be added together. For each counter the number of cycles is $\sigma+2s$

where σ is the length of the counter and s is the length of the sum. The sum will have σ bits at the start and have $\sigma + \log_2 A$ bits at the end, where A is the number of active location addresses. If $\sigma = 8$ and $A = 160$ we get $s = 8$ at the start and $s = 16$ at the end, and maximally $8 + 2(16) = 40$ cycles per counter are needed. This gives a total maximum below 5500 ($\approx 138 * 40$).

6. TIMING OF SDM USING REMAP

Using the figures in Section 5, we now estimate the performance for two different configurations (with or without counters) of REMAP on two differently sized problems. We assume a transposed mapping is used, and that a select first network (cf. Section 4.3.1) is used. For the updates/s figures a 20 MHz clock is assumed. There is some control unit overhead for things like address calculation and subroutine calls which is *not* included in the figures above. This overhead could be estimated to be between 20 and 50% for a typical controller.

- The small model (α=256, δ=256, N=8192, σ=8, r=111 => $\bar{A}$=160) using 256 PEs.

	With counter	*Without counter*
Separate phases		
Selection	17152	58240
Store	5696	5696
Retrieve	<9600	<9600
Total		
Storing		
Total cycles	22900	63900
Updates/s	**875**	**310**
Retrieving		
Total cycles	<27000	<68000
Updates/s	**740**	**290**

- The large model (α=1024, δ=1024, N=2^{20}, σ=12, r=451 => $\bar{A}$=1124) using 1024 PEs.

	With counter	*Without counter*
Separate phases		
Selection	2.1M	8.0M
Store	53k	53k
Retrieve	<89k	<89k
Total		
Storing		
Total cycles	2.2M	8.0M
Updates/s	**9.0**	**2.5**
Retrieving		
Total cycles	<2,2M	<8.0M
Updates/s	**9.0**	**2.5**

For very large SDM models, like the latter model above, the selection phase is totally dominating and adding new PE arrays as suggested in the end of Section 4.3 would have a large positive impact on the computation time. For the first hundred arrays an almost linear speed-up could be expected. As an example, using 16 modules of SIMD arrays the performance on the above model would be well above 100 cycles per second (using Hamming distance counters).

7. STORAGE REQUIREMENTS

The storage requirements will depend on the number of address bits, α, data bits, δ, location addresses (physical addresses), N, and the size of the counters, σ. The total memory requirements (in bits), M, can then be expressed as: $M = \alpha N + \sigma\delta N$. Having n PEs we need $m = M/n$ bits of memory per PE.

For very small SDM models (α=256, δ=256, N=8192) both total and per PE memory requirements cause no problems, but for the large model (α=1024, δ=1024, N=2^{20}) there are few computers with the required amount of memory (2 - 4 GByte). If the number of PEs is small, a very large amount of memory is needed in each PE (20-40 Mbit). If we instead have many modules each bit-serial PE can have a normal amount of memory for typical applications (e.g. 1-4 Mbits). It is again worth noting that the use of modules also increases the general performance of the architecture.

8. OTHER IMPLEMENTATIONS

The SDM algorithm has been implemented both on commercial machines and some specially built hardware. A discussion on their performance and suitability for SDM calculations follows below.

8.1 Connection Machine (CM, CM-2)

The Connection Machine [11, 39], manufactured by Thinking Machines Corporation, is for the moment the most massively parallel machine built (8k up to 64k processing elements). Its strong side is the powerful hypercube connection and the sheer number of processors. Besides the hypercube there is a mesh connection network. For the second generation (CM-2) floating point support was added, arranged as one FP unit per 32 PEs. This means 2048 FP units on a 64k machine, giving a total of 20 Gigaflops.

Rogers [29] has used CM-2 as a workbench for exploring Kanerva's SDM model. For the selection phase he used rowwise mapping. But for the store/retrieve phase he used weight parallelism (as many PEs as there are counters). As the physical number of PEs was of the same order as one column of counters he implicitly used node parallelism and rowwise mapping, letting the CM-2 sequencer take care of the looping over each column. Implementing this in *Lisp gave a performance of only ≈3 iterations per second (α=256, δ=256, N=8192). Using a pure rowwise mapping in C* the present author has been able to achieve between 30 and 70 iterations per second. The difference is probably due to some unnecessary but expensive communication needed going from 1D to 2D representation.

It should be noted that floating point values should be used for the counters using the rowwise mapping of SDM onto a CM-2. This seems inefficient at first but for the retrieve phase when a summation across the PEs (sum-reduction) is carried out, the performance using a float is *eight* times better than when using integers. It seems that the way the routers can utilize the floating point chips doing sum-reduction will make it more efficient to use floating point counters instead of integer counters!

For the CM-2 the transposed mapping is hard to use as the conditional processing must be done in the host (as one can not program the sequencer to do that, or at least it is not very well documented), and the communication between the processor array and the host is not very efficient. Neither is there any support to do addition and subtraction at the same time in different PEs.

8.2 Hardware Implementations

STANFORD

Flynn et al. have, in a collaboration between RIACS and Stanford University, designed a prototype for SDM [9]. They chose to use columnwise mapping and pipelining between the two phases (select and store/retrieve). An ordinary MC68030 computer is used for the control module that implements a SCSI interface for the prototype. No identifiable processors are used in the module for the select phase, instead an adder-tree together with a comparator performs all the needed computation. It is realized using PLAs and EPROMs. The up/down counters are contained in a separate module operated by another MC68030. The design can use up to 256-bit addresses and 8k to 128k location addresses. Using 8k memory locations it can run approximately 50 cycles per second.

TAMPERE

Saarinen et al. [32, 33] have suggested a hardware implementation of SDM using the work of Flynn et al. [9] as a starting point. Like Flynn et al. they have chosen columnwise mapping. The address unit is very similar to that of the Stanford prototype, but newer technology like FPGA is used to realize the adder-tree and the other logic needed for the search phase. An analog address comparator based on summation of currents is also suggested. The counters used in the second phase are operated by FPGAs and are based on banks of dynamic RAMs. The counters are accessed 8 or 4 in parallel. To be able to experiment with modification of the algorithm, the up/down counters are replaced with adder/subtractors of 8-bit values.

The performance is not stated exactly, but Lindell [20] estimate their hardware to run about 230 iterations/s for write and similar figures for read (α=256, δ=256, N=8192, σ=8, r=111 => $\bar{A}$=160).

9. CONCLUSIONS

The main idea of the SDM implementation suggested in this paper is to use a mapping onto a linear array which can utilize as many PEs as possible in each step. This can be achieved by using rowwise mapping for the selection phase and columnwise mapping for the store/retrieve phase. Each processing element can be very simple and bit-serial architectures are especially suitable for the otherwise time consuming selection phase. By the addition of a counter to each PE, the selection time can be reduced by a factor of three to four.

Timing calculations using a PE with counters indicate that it is possible to run SDM at speeds 10-30 times that of an 8k CM-2. This is accomplished with only 256 REMAP PEs. If we normalize with respect to the clock frequency and number of PEs, the measure of speed would be 150 times that of CM-2 per PE. The relatively poor performance on CM-2 is found to depend on at least three factors: underutilization of PEs during the select/retrieve phase; the natural rowwise mapping demands time consuming sum-reduction across PEs; the "optimal" transposed mapping is hard to implement efficiently with the current sequencer.

In our study, the multi-modular concept is found to be a better way to increase the degree of parallelism then just adding PEs to an ordinary SIMD array.

Communication is simple in SDM as in many other artificial neural network algorithms [24, 37]. The form of communication needed is found in many other, conventional algorithms (broadcast, nearest neighbor, select/first) and may be included at small cost in logic and wire.

Also when comparing our REMAP computer to direct hardware implementations of SDM, like the one described in Flynn et. al. [9], our approach seems very promising. The first small (128 PEs) version of the REMAP computer was completed during 1993. Unfortunately the FPGA reprogramming tools have made it difficult to experiment with the architecture as intended. Thus, we have yet to implement a version containing counters instead of multipliers.

10. REFERENCES

[1] Albus, J. S., *Brains, Behavior, and Robotics*, Petersborough, NH, USA: BYTE/McGraw-Hill, 1981.

[2] Arbib, M. A., "Schemas and neural network for sixth generation computing," *Journal of Parallel and Distributed Computing*, vol. 6, no. 2, pp. 185-216, 1989.

[3] Bengtsson, L., A. Linde, B. Svensson, M. Taveniku and A. Åhlander, "The REMAP massively parallel computer platform for neural computations," in *Third International Conference on Microelectronics for Neural Networks (MicroNeuro '93)*, Edinburgh, Scotland, UK, pp. 47-62, 1993.

[4] Blevins, D. W., E. W. Davis, R. A. Heaton and J. H. Reif, "Blitzen: A highly integrated massively parallel machine," *Journal of Parallel and Distributed Computing*, vol. 8, pp. 150-160, 1990.

[5] Clarke, T. J. W., R. W. Prager and F. Fallside, "The modified Kanerva model: Theory and results for real-time word recognition," *IEE Proceedings-F*, vol. Vol 138, no. 1, pp. 25-31, 1991.

[6] Danforth, D. G., "An empirical investigation of sparse distributed memory using discrete speech recognition," in *INNC 90 Paris. International Neural Network Conference*, Paris, France, vol. 1, pp. 183-186, 1990.

[7] Davis, E. W., T. Nordström and B. Svensson, "Issues and applications driving research in non-conforming massively parallel processors," in *Proceedings of the New Frontiers, a Workshop of Future Direction of Massively Parallel Processing*, I. D. Scherson Ed., McLean, Virginia, pp. 68-78, 1992.

[8] Fernström, C., I. Kruzela and B. Svensson, *LUCAS Associative Array Processor - Design, Programming and Application Studies*, vol. 216 of *Lecture Notes in Computer Science*, Berlin: Springer Verlag, 1986.

[9] Flynn, M. J., P. Kanerva and N. Bhadkamkar, "Sparse distributed memory: Priciples and operation," Tech. Rep. 89.53 RIACS, NASA Ames Research Center, Moffet Field, CA, 1989.

[10] Hillis, W. D., *The Connection Machine*, Cambridge, Massachusetts: The MIT Press, 1985.

[11] Hillis, W. D. and G. L. J. Steel, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170-1183, 1986.

[12] Kanerva, P., "Adjusting to variations in tempo in sequence recognition," in *Neural Networks Supplement: INNS Abstracts*, vol. 1, pp. 106, 1988.

[13] Kanerva, P., *Sparse Distributed Memory*, Cambridge, MA: MIT press, 1988.

[14] Kanerva, P., "A cerebellar-model associative memory as a generalized random-access memory," in *Digest of Papers. COMPCON Spring '89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*, San Francisco, CA, USA, pp. 570-576, 1989.

[15] Kanerva, P., "Contour-Map Encoding of Shape for Early Vision," Tech. Rep. 90.5 RIACS, NASA Ames Research Center, 1990.

[16] Kanerva, P., "Efficient packing of patterns in sparse distributed memory by selective weighting of input bits," in *Proceedings of the 1991 International Conference Artificial Neural Networks*, T. Kohonen, et al. Eds. Espoo, Finland, vol. 1, pp. 279-284, 1991.

[17] Keeler, J. D., "Capacity for patterns and sequences in Kanerva's SDM as compared to other associative memory models," in *Neural Information Processing Systems*, D. Z. Anderson Ed., Denver, CO, 1987.

[18] Kohonen, T., "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464-1480, 1990.

[19] Linde, A., T. Nordström and M. Taveniku, “Using FPGAs to implement a reconfigurable highly parallel computer,” *Field-Programmable Gate Array: Architectures and Tools for Rapid Prototyping; Selected papers from: Second International Workshop on Field-Programmable Logic and Applications (FPL'92), Vienna, Austria*, H. Grünbacher and R. W. Hartenstein Eds. New York: Springer-Verlag, pp. 199-210, 1992.

[20] Lindell, M., “Hardware realization of sparse distributed memory,” Technical Report 12-93, Tampere University of Technology, Finland, 1993.

[21] Mackie, S. and J. S. Denker, “A digital implementation of a best match classifier,” in *IEEE 1988 Custom integrated Curcuits Conference*, Rochester, NY, pp. 10.4.1-10.4.4, 1988.

[22] Nordström, T., “On-line spatially localized learning systems, part I - model description,” *to be submitted (Also available as Res. Rep. TULEA 1995:1, Luleå University of Technology, Sweden)*, 1995.

[23] Nordström, T., “On-line spatially localized learning systems, part II - parallel computer implementation,” *to be submitted (Also available as Res. Rep. TULEA 1995:2, Luleå University of Technology, Sweden)*, 1995.

[24] Nordström, T. and B. Svensson, “Using and designing massively parallel computers for artificial neural networks,” Res. Rep. TULEA 1991:13, Luleå University of Technology, Sweden, 1991.

[25] Poggio, T. and F. Girosi, “A theory of networks for approximation and learning,” A.I. Memo 1140 (first released 1991), Massachusetts Institute of Technology, 1994.

[26] Powell, M. J. D., “Radial basis functions for multivariable interpolation: a review,” in *IMA Conference on Algorithms for the Approximation of Functions and Data*, RMCS, Shrivenham, UK, pp. 143-167, 1985.

[27] Prager, R. W., T. J. W. Clarke and F. Fallside, “The modified Kanerva model: results for real time word recognition,” in *First IEE International Conference on Artificial Neural Networks*, London, UK, pp. 105, 1989.

[28] Prager, R. W. and F. Fallside, “The modified Kanerva model for automatic speech recognition,” *Computer Speech and Language*, vol. 3, pp. 61-81, 1989.

[29] Rogers, D., “Kanerva's sparse distributed memory: an assosiative memory algorithm well-suited to the connection machine,” Tech. Rep. 88.32 RIACS, NASA Ames Research Center, 1988.

[30] Rogers, D., “Kanerva's sparse distributed memory: an assosiative memory algorithm well-suited to the connection machine,” in *Proceedings of the Conference on Scientific Application of the Connection Machine*, Moffet Field, Ca, USA, vol. 1, pp. 282-298, 1988.

[31] Rogers, D., “Predicting weather using a genetic memory: a combination of Kanerva's sparse distributed memory with Holland's genetic algorithms,” in *Neural Information Processing Systems 2*, D. Touretzky Ed., Denver, CO, pp. 455-464, 1989.

[32] Saarinen, J., P. Kotilainen and K. Kaski, “VLSI architectures of sparse distributed memory,” in *1991 IEEE International Sympoisum on Circuits and Systems*, Singapore, vol. 5, pp. 3074-3077, 1991.

[33] Saarinen, J., M. Lindell, P. Kotilainen, J. Tomberg, P. Kanerva and K. Kaski, “Highly parallel hardware implementations of sparse distributed memory,” in *Artificial Neural Networks. Proceedings of the 1991 International Conference. ICANN-91*, Espoo, Finland, vol. 1, pp. 673-678, 1991.

[34] Saarinen, J., S. Pohja and K. Kaski, “Self-organization with Kanerva's sparse distributed memory,” in *Artificial Neural Networks. Proceedings of the 1991 International Conference. ICANN-91*, T. Kohonen, et al. Eds. Espoo, Finland, vol. 1, pp. 285-290, 1991.

[35] Specht, D. F., “Probabilistic Neural Networks,” *Neural Networks*, vol. 3, no. 1, pp. 109-118, 1990.

[36] Sutton, R. S. and S. D. Whitehead, “Online learning with random representations,” *To appear*, 1993.

[37] Svensson, B. and T. Nordström, “Execution of neural network algorithms on an array of bit-serial processors,” in *10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*, Atlantic City, NJ, USA, vol. II, pp. 501-505, 1990.

[38] Svensson, B., T. Nordström, K. Nilsson and P.-A. Wiberg, “Towards modular, massively parallel neural computers,” *Connectionism in a Broad Perspective: Selected Papers from the Swedish Conference on Connectionism - 1992*, L. F. Niklasson and M. B. Bodén Eds. Ellis Horwood, pp. 213-226, 1994.

[39] Thinking Machines Corporation, “Connection Machine, Model CM-2 technical summary,” Version 5.1 T M C Cambridge, Massachusetts, 1989.

[40] Vanhala, J. and K. Kaski, “Simulating neural networks in distributed environments,” Res. Rep. 6-89 Department of Electrical Engineering, Electronics Laboratory, Tampere University of Technology, 1989.

Presented at: DSA-92
Fourth Swedish Workshop on Computer System Architecture
Linköping, January 13th-15th, 1992

Designing Parallel Computers for Self Organizing Maps

Tomas Nordström

Division of Computer Science & Engineering
Department of Systems Engineering and Mathematics
Luleå University of Technology, Sweden
E-mail: tono@sm.luth.se

ABSTRACT

Self organizing maps (SOM) are a class of artificial neural network (ANN) models developed by Kohonen. There are a number of variants, where the self organizing feature map (SOFM) is one of the most used ANN models with unsupervised learning. Learning vector quantifiers (LVQ) is another group of SOM which can be used as very efficient classifiers. SOM have been used in a variety of fields, e.g. robotics, telecommunication and speech recognition.

Currently there is a great interest in using parallel computers for ANN models. In this report we describe different ways to implement SOM on parallel computers. We study the design of massively parallel computers, especially computers with simple processing elements, used for SOM calculations.

It is found that SOM (like many other ANN models) demands very little of a parallel computer. If support for broadcast and multiplication is included very good performance can be achieved on otherwise modest hardware.

1.0 INTRODUCTION

The algorithms we study in this report are Kohonen's self organizing maps (SOM) and variants of them. These maps have been used in pattern recognition, especially in speech recognition [27], but also in robotics and automatic control [40, 46] and telecommunication tasks [3, 32]. This study is part of a series of reports [43, 44, 49] that shows how well suited bit-serial SIMD computers are for simulating artificial neural networks.

As an example of bit-serial SIMD computers, REMAP^3 (reconfigurable, embedded, massively parallel processor project) will be used. As the processing elements are reconfigurable it is possible to include different types of support for different kinds of algorithms. For back-propagation [47] and Hopfield networks [18, 19, 20] a bit-serial multiplier has been found to be essential for the performance [44, 49]. For the implementation of Kanerva's SDM model [25] the multiplier was not needed, instead a counter was suggested [43]. In this report we try to recognize architectural principles and components that are essential for the efficient calculation of Kohonen's models.

In the next section we describe the background of SOM. After that, two sections discuss implementation considerations and ways to map SOM onto a computer architecture. Then follows a section where some of the existing parallel implementations are discussed. Finally, we draw some conclusions concerning the task of designing parallel computers for SOM.

2.0 BACKGROUND

An overview of the different models of self organizing maps and the application areas where they have been used can be found in [26, 28, 29, 30, 31]. Below we only restate the basic models and refer to the references above for more details.

2.1 Competitive Learning

In competitive learning [30, 47] the responses from the adaptive nodes (weight vectors) tend to become localized. After appropriate training the nodes specify clusters or codebook vectors that approximate the probability density functions of the input vectors. Algorithm 1 is an example of a competitive learning algorithm. If the spatial relationships of the resulting feature sensitive nodes are not considered we get a zero-order topology map.

Algorithm 1 Competitive learning (zero-order topology).

1. Find the node (or weight vector) w_i closest to input x.
$$\|x(t_k) - w_c(t_k)\| = \min_i \|x(t_k) - w_i(t_k)\|$$
2. Make the winning node closer to input.
Where $i = c$
$$w_i(t_{k+1}) = w_i(t_k) + \alpha(t_k)\,[x(t_k) - w_i(t_k)]$$
otherwise
$$w_i(t_{k+1}) = w_i(t_k)$$
3. Repeat from step 1 while reducing the learning rate α.

2.1.1 Adding Conscience

A problem with the algorithm above is that instead of placing the nodes according to the input point density function $p(x)$ the nodes are placed as $p(x)^{M/(M+2)}$. Having low dimensional input vectors (i.e. small M) there will be a bias towards the low probability regions. DeSieno [6] has found that adding conscience to the competitive learning algorithm will greatly improve the encoding produced by the map. The idea is that the nodes should be conscientious about how many times they have won, compared to other nodes, see Algorithm 1. That is, every node should win the competition approximately the same

number of times. Another way to improve the clustering is to use a higher-order topology map, like the Kohonen model, this is especially true for non-continuous input probability density functions.

Algorithm 2 Competitive learning with conscience (zero-order topology).

1. Find the node (or weight vector) w_i closest to input x using a conscience c_i as offset (C is a scaling factor).
$$\|x(t_k) - w_c(t_k)\| = \min_i (\|x(t_k) - w_i(t_k)\| + Cc_i)$$
2. Make the winning node closer to input.
Where $i = c$
$$w_i(t_{k+1}) = w_i(t_k) + \alpha(t_k)\,[x(t_k) - w_i(t_k)]$$
otherwise
$$w_i(t_{k+1}) = w_i(t_k)$$
3. Repeat from step 1 while reducing the learning rate α, and increasing the conscience of the winning neuron $c_i(t_{k+1}) = c_i(t_k) + 1$

2.2 Kohonen Learning

In the brain there are many areas, such as the visual and somatosensory cortex, which are organized in a way that reflects the organization of the physical signals stimulating the areas i.e. they are topological maps.

Inspired by that, Kohonen has developed a class of artificial neural network (ANN) models which develop these, so called, self organizing maps (SOM), also referred to as topological feature maps (TFM). They are all models with competitive learning and use first, second, or higher order topological maps [26].

SOM may be formed with unsupervised learning, i.e. without any teacher saying what is right or wrong. This type of SOM is referred to as self organizing feature maps (SOFM), see Algorithm 3.

Algorithm 3 The SOFM algorithm (higher-order topology)

1. Find the node (or weight vector) w_i closest to input x.
$$\|x(t_k) - w_c(t_k)\| = \min_i \|x(t_k) - w_i(t_k)\|$$
2. Find the neighbourhood $N_c(t_k)$.
3. Make the nodes in the neighbourhood closer to input.
Where $i \in N_c(t_k)$
$$w_i(t_{k+1}) = w_i(t_k) + \alpha(t_k)\,[x(t_k) - w_i(t_k)]$$
otherwise
$$w_i(t_{k+1}) = w_i(t_k)$$
4. Repeat from step 1 with ever decreasing neighbourhood N_c and gain sequence α ($0 < \alpha < 1$).

If the resulting maps are to be used as classifiers, and the training example classes are known (i.e. supervised learning), a fine tuning of the SOFM model called learning vector quantization (LVQ) model has been suggested [29, 30]. In the simplest version the difference is that, in Step 3 in Algorithm 3, $\alpha(t_k)$ is negated if w_i belongs to the wrong class. No neighbourhood is used for LVQ.

2.3 Stochastic Competitive Kohonen Learning

Van den Bout and Miller III [52, 53] have suggested a modification to competitive and Kohonen learning which simplifies the calculations by replacing the "analog" signals with stochastic binary signals. In their model, called TInMANN, the mean value of a stochastic binary signal is viewed as an analog signal in the range [0,1]. This signal representation leads to very simple (and space efficient) digital logic for the computations needed in the algorithms. For example, multiplication of stochastic signals can be computed using only a simple AND gate.

By noting that the weight vectors slowly integrate the effects of the environmental stimuli, this can be done by incrementing/decrementing the weights stochastically with a probability proportional to the strength of the input vector. Obviously, more iterations are needed, but as each step is computationally very simple the overall computation time will decrease, see section 5.8.

3.0 IMPLEMENTATION CONSIDERATIONS

SOM models have been implemented on a large number of different parallel computers: Warp [39], Transputers [17, 45, 48], Connection Machine [45], MasPar [11] and in special hardware [8, 24, 37, 38, 52, 53].

Aspects that have to be considered when implementing SOM on parallel computers are: communication facilities, computational capabilities, mapping and partitioning of the algorithm. Some architectures benefit a great deal if floating point numbers can be avoided, and the integer precision needed for the weights should be analysed.

The algorithms in section 2.0 can be divided into four steps:

1. Finding the distance from the input to the nodes.
2. Finding the node closest to input.
3. Determining the neighbourhood to the closest node.
4. Updating the neighbourhood nodes.

The rest of this section will discuss the different aspects of implementing SOM based on these four steps.

3.1 Communication

The weight vectors can be assumed to be distributed over the processing elements (PEs). Thus the first step requires the input vector to be distributed. The most effective way to do this is by utilizing broadcast. If the architecture does not have broadcast, for example if Transputers are going to be used, a nearest neighbour communication like ring or mesh must be used.

For the second step a minimum distance must be found. A minimum could be found by doing N–1 comparisons among the N nodes. Some "global" or "token-ring" communication is needed if the nodes are distributed over many PEs. As seen in section 4.4 a very fast bit-serial method exists to find min/max among PEs if a global-or function exists.

The activation/selection of the neighbours can either be solved as a spatial distance calculation or as a nearest neighbour communication. The time for communication will depend on the neighbourhood size. When the neighbourhood size is small the communication method can be faster than spatial distance calculation. The topology of the computer should support the communication needed, which depends on the spatial topology of the SOM model.

If the input vector (or the difference between the node and input) is stored no communication is needed during step 3.

3.1.1 Summary on Communication

For the basic SOM models broadcast is the most efficient way of communication.

By computing which nodes to be considered as neighbours, instead of using nearest neighbour communication, the computer topology becomes irrelevant for the algorithm. It also makes the time for step 3 constant with respect to the neighbourhood size.

3.2 Computations

For all of the algorithms in section 2.0 the distance is calculated using a Euclidean metric. Another much used metric is the dot-product metric. Both metrics are discussed in this section.

Some model parameters used in this and following sections are:

- N The number of nodes
- N_c Size of the neighbourhood (may depend on time)
- M The dimension of the input vector
- δ Bits used for weights
- n The number of processing elements available.

3.2.1 Euclidean Metric

Using a Euclidean distance metric in the first step means that the distance $\|x(t_k) - w_i(t_k)\|$ is calculated as

$$\sqrt{\sum_j (x_j(t_k) - w_{ij}(t_k))^2}$$

Note that the square-root function does not need to be evaluated, as the square-root is a monotonic function and the result is used for comparison only.

We get the following estimate on the number of computations needed in each step:

1. We must do NM subtractions, NM squarings and NM additions. Note that the result of the subtraction can be reused in step 3 if there is enough memory to store the result.
2. The number of calculations needed for finding minimum is N–1 comparisons (subtractions).
3. If we instead of communicating, calculate the neighbourhood, it should be possible to calculate the spatial distance in less than 3 operations (subtract, square, add) per spatial dimension (usually 1-3).
4. For the updating part we must carry out (N_cM subtractions,)N_cM multiplications andN_cM additions. The subtraction can be carried out in step 1 if there is enough memory to store the result.

The total number of operations is (assuming two spatial dimensions and recalculation of the subtraction)

$$O = 3MN + (N-1) + 6 + 3MN_c.$$

Using a reasonable value forN_c like$N/4$ we get approximately

$$O \approx (1 + 3.75M)\, N \text{ operations per training example.}$$

3.2.2 Dot-Product Metric

By normalizing the input vector (i.e. keeping $\|x(t_k)\| = 1$), and keeping the nodes normalized after updating, the dot-product can be used instead of sum of squares. The distance calculation can then be calculated as a “weight matrix times an input vector” like in many other neural network models.

The matching law is modified to

$$x(t_k)^T w_c(t_k) = \max_i \{ x(t_k)^T w_i(t_k) \}$$

The update law must then be modified to

$$\text{where } i \in N_c(t_k) \qquad w_i(t_{k+1}) = \frac{w_i(t_k) + \alpha'(t_k)\, x(t_k)}{\left\| w_i(t_k) + \alpha'(t_k)\, x(t_k) \right\|}$$

$$\text{otherwise} \qquad w_i(t_{k+1}) = w_i(t_k)$$

Stillα' is a monotonically decreasing function, but now $0 < \alpha' < \infty$. The neighbourhood is the same decreasingN_c as before.

We get the following estimate on the number of computations needed in each step:

1. For the dot-product metric the NM subtraction, used by Euclidean distance metric, can be avoided. The squarings are replaced by multiplications.
2. Same as Euclidean metric, but we want to find maximum instead of minimum.
3. Same as Euclidean metric.
4. Using dot-product distance metric we must doN_cM multiplications andN_cM additions for the nominator. For the denominator we must use an additionalN_cM squarings andN_cM additions. We must also do one division.

The total number of operations is (assuming two spatial dimensions)

$$O = 2MN + (N-1) + 6 + 4MN_c + 1.$$

Using a reasonable value forN_c like$N/4$ we get approximately

$$O \approx (1 + 4M)\, N \text{ operations per training example.}$$

The dot-product calculation can in some technologies be very fast (e.g. optical transmission filters). But for our purpose the overhead for the normalization is too time consuming. This will be accentuated if SIMD computers are to be used.

3.2.3 Precision Used for the Weights

If we intend to use a bit-serial architecture, where each PE only operates on one bit at a time, the precision becomes very important. An analysis made by J. Mann [37] shows that at least 8 bits seems necessary for the weights. He also finds that Euclidean distance measures are not as sensitive as dot-product metric to weight precision.

It also seems that the updating rule could be changed to an increment/decrement of the weight, depending on the sign of the difference between the weight and the input without much influence on the performance of the SOM algorithm.

Hammerstrom and Nguyen have found the SOM to be sensitive to error bias from bit truncation, more sensitive than to average quantization error from reduced precision. They also reduce the error accumulation using saturation arithmetic (on overflow the max/min value is used).

3.2.4 Summary and Comments on Computation

Almost half of the operations computed are multiplications, and therefore support for multiplication is very important. As

calculations using short fixed-point numbers seem possible, bit-serial computers become interesting. The Euclidean metric uses less operations for its computation, and is less sensitive to weight quantization, and is therefore more attractive to use than dot-product.

Many have used CUPS (connection updates per second) as a measure of performance on SOM algorithms. It has been used as the number of connections *MN* multiplied by the number of updates per second *u* i.e *MNu*. This despite the fact that very few connections are actually updated when using small neighbourhoods.

Even if the word *updates* is misleading, the CUPS measure, used in relation to the FLOPS (floating point operations per second) or IPS (instructions per second) measures, can be used as an indication of the efficiency of the architecture (on SOM algorithms). We may define an efficiency measure like:

$$E = \frac{\text{operations per second}}{\text{maximum number of operations}} = \frac{Ou}{\text{IPS}_{\max}} \approx \left(\frac{3.75MNu}{\text{IPS}_{\max}} = 3.75\frac{\text{CUPS}}{\text{IPS}_{\max}} \right) \quad \textbf{(EQ 1)}$$

Where $\text{IPS}_{\max}$ is the maximum number of operations per second achievable on the computer. The corresponding measure for floating point calculations is:

$$E = 375\frac{\text{CUPS}}{\text{FLOPS}_{\max}}$$

4.0 MAPPING SOM ONTO A COMPUTER ARCHITECTURE

4.1 Computer Architectures

One of the most used divisions of architectures is due to Flynn [10]. He divided the computers into groups according to the number of instruction streams and the number of data streams:

1. SISD - Single Instruction stream, Single Data stream.
2. SIMD - Single Instruction stream, Multiple Data streams.
3. MISD - Multiple Instruction streams, Single Data stream. (Anomaly of the division)
4. MIMD - Multiple Instruction streams, Multiple Data streams.

Single instruction stream multiple data stream (SIMD) computers is a computer model where the same instruction is executed in each of the processing elements (PEs). Using this model it is possible to achieve massive parallelism at small cost. It is found that almost all ANN algorithms fit very well into the SIMD model [44]. The SOM algorithms in section 2.0 also seem to map easily onto the SIMD computer model. This hypothesis will be found to be true later in this section. Below there follows a discussion on how to implement SOM on a SIMD computer.

4.2 Degree of Parallelism

Looking at an ANN algorithm such as SOM there are (at least) six different ways of achieving parallelism [44]. The typical degree of parallelism varies widely between the six different kinds, as the table below shows.

Parallelism	Typical range SOM:
Training session	$10 - 10^3$
Training example	$10 - 10^7$
Forward-Backward	1 - 2
Node (neuron)	$100 - 10^6$
Weight (synapse)	$2 - 10^4$
Bit	1 - 64

To utilize the computing resources of a massively parallel computer (thousands of processing elements) efficiently the table indicates that we must use at least one of the following dimensions:

Training session parallelism.
Training example parallelism.
Node parallelism.
Weight parallelism.

Note that the first two dimensions of parallelism are of interest only in batch processing situations, i.e. when training the network. If the network is to be to used in a real-time situation, interacting with the outside world, training session and training example parallelism would be unavailable. In those cases, node and/or weight parallelism must be chosen, possibly in combination with e.g. bit and layer parallelism.

4.3 Node Parallelism on SIMD Computers

Unfortunately the amount of node parallelism used for updating the nodes varies with the size of neighbourhood. Still the node parallelism is the most used and maybe the most natural mapping for SIMD computers. The mapping may be visualized for a 1D map topology on a linear processor array as in Figure 1.

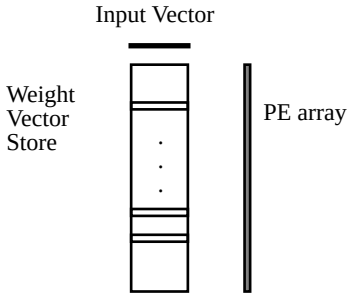

Figure 1 Using node parallelism to map SOM onto a linear processor array.

For the four steps we then get the following estimate of the number of computation steps needed, using Euclidean distance metric and having the same number of processing elements as the number of nodes, i.e. $n = N$.

1. Calculating the distance requires *M* subtraction steps, *M* squaring steps and *M* addition steps. The result of the subtraction can be reused in step 4 if there is enough memory to store the result.
2. The number of calculations needed for finding the minimum distance varies with the communication structure. Using bit-serial arithmetic and global-or function, as later described in section 4.4, the comparison can be computed in less than $2(2\delta + \log_2 M)$ (Two times the number of bits in the sum of squares) steps.

3. The time to determine the neighbourhood varies with the communication structure, but it should always be possible to calculate in less than 3 operations per spatial dimension.
4. Updating the weights requires (M subtractions), M multiplications and M additions. Note that the efficiency during this step is only N_c / N i.e. the efficiency will go down as the neighbourhood shrinks.

The total number of steps is, using a 2D map as an example, approximately

$$O = 3M + 2(2\delta + \log_2 M) + 6 + 3M$$

4.4 Bit-serial SIMD Implementation

The majority of massively parallel processors use bit-serial arithmetic, and that is also the basic mode of operation for our own research machine, REMAP[3]. Therefore, we would like to analyse the algorithms down to bit-level. For the majority of operations using bit-serial PEs, the processing times grow linearly with the data length used. E.g. the time to do a bit-serial addition is the same time as to read the operands and store the result (3δ cycles). This may be regarded as a serious disadvantage (e.g when using 32- or 64-bit floating point numbers), or as an attractive feature (use of low precision data speeds up the computations accordingly). In any case, bit-serial data paths simplify communication in massively parallel computers.

As concluded in section 3.2.4 support for multiplication is important. Unfortunately, very few bit-serial computers have support for bit-serial multiplication, and without such, multiplication time grows quadratically with the data length. However, with an inclusion of a bit-serial multiplier [9, 44, 55] the multiplication of two δ bit numbers can be performed in 4δ cycles (i.e. the time to read the operands and store the result).

Using the natural mapping for SIMD computers (node parallelism) and having $n = N$ we get the following number of cycles for the SOM algorithm using Euclidean metric.

1. We have to do $3\delta M$ cycles for subtraction, $4\delta M$ cycles for squaring and $6\delta M$ cycles for addition. If the result from the multiplier is used directly for addition (i.e. without storing the result in memory) the total time for squaring and adding will be $8\delta M$.
2. Finding minimum (or maximum) using bit-serial working mode can be implemented very efficiently. A global-or function is needed to check a bit "slice" if all bits are equal, also the means to turn off PEs depending on the result of the previous operation are needed. Assuming we want to find minimum (i.e. using Euclidean distance metric), the search starts by examining the most significant bit of each value. If anyone has a zero, all PEs with a one are turned off (otherwise we restore the previous state). The search goes on in the next position, and so on, until all bit positions have been treated. The time for this search is independent of the number of values compared, it depends only on the data length. The maximal number of cycles needed will be two times the length of the data i.e $2(2\delta + \log_2 M)$. To resolve multiple minimums a select first network [9] can be used to select the first active processor.
3. Wanting to have constant time for this step we want to calculate the neighbourhood (instead of communicate it). There are a number of variants depending on the (spatial) distance measure and the map topology. If 7 bits are used for the spatial coordinates, the following estimates for the cycle count can be given:

- Euclidean (spatial) distance

  A second order topology map can be computed in 206 cycles

- City block (spatial) distance

  A first order topology map can be computed in 48 cycles

  A second order topology map can be computed in 96 cycles

4. Updating of the weight vectors can be computed with $3\delta M$ cycles for subtraction, $4\delta M$ cycles for multiplication and $3\delta M$ cycles for addition. If the global constant α is loaded in parallel from the controller, and the result from the multiplier is used directly for the addition, the number of cycles in step 4 will be $7\delta M$.

There are of course some cycles needed for overflow tests and initiations but the above indicates that SOM could be calculated in typically: $18\delta M + 250$ cycles. Note, that we assume $n = N$. Using the approximate total number of operation $O \approx (1 + 3.75M)N$, we can calculate the efficiency $E = (Ou) / \mathrm{IPS}_{\max}$.

Let C stand for the clock frequency. As the time for multiplication is 4δ the $\mathrm{IPS}_{\max}$ is $(Cn) / (4\delta)$. The update rate will be $u = (Cn) / (18\delta MN + 250N)$ and the efficiency

$$E = \frac{4(1 + 3.75M)\delta}{18\delta M + 250}$$

The asymptotic efficiency will be 83%. Already for M=10 and δ=8 the efficiency is 73%. These figures show that a bit-serial SIMD computer can be used very efficiently for SOM calculations.

4.5 Other Forms of Parallelism

If the input vectors are long (i.e. large M) and the neighbourhood N_c is small, a "transposed" mapping could be considered. This is the same as the columnwise mapping suggested for SDM [43]. For SOM this would correspond to weight parallelism with iteration over the nodes, see Figure 2.

Using this mapping the summation of squares must be done across the PEs, and special hardware (e.g. an adder-tree) should be included for efficiency reasons. However, with this hardware available it is possible to pipeline the addition and comparison (carried out in the controller) with the subtraction and squaring (carried out in the processor array).

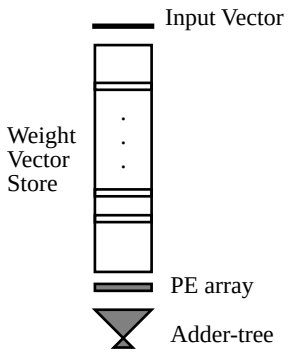

Figure 2 Using weight parallelism to map SOM onto a linear processor

For the last step, weight parallelism is faster than node parallelism with a factor M/N_c. This will be a considerable factor when N_c is small and M is large. The second and third steps are almost "for free" for weight parallelism as pipelining can be used. As the number of nodes N often is larger than the number of elements M in the input vector, and for weight parallelism we only have $n = M$ PEs, this mapping will be less efficient than node parallelism during the first step. The factor is $(3M)/(2N)$.

It seems that the most efficient mapping during the first step is node parallelism and during the fourth step weight parallelism (as for the SDM model [43]). But we have not found any way to achieve mixed mapping for SOM onto any normal SIMD computer.

Note that if $2N > 5M$, no matter how fast the three last steps are when weight parallelism is used, it will still be less efficient than the node parallel version. This is because its first step takes longer than the total time of the node parallel version.

5.0 IMPLEMENTATIONS ON PARALLEL COMPUTERS

In the following subsections many of the parallel computers used for SOM are described. Performance figures for the SOM algorithm on these computers are also given if they are available.

A summary of the discussed implementations is shown in Figure 3.

5.1 CNAPS

CNAPS (Connected Network of Adaptive ProcessorS) manufactured by Adaptive Solutions is one of the first architectures developed especially for ANN. It was called X1 in the first description by Hammerstrom [14]. It is a 256 PE SIMD machine with a broadcast interconnection scheme. Each PE has a multiply (9x16bit) and add arithmetic unit and a logic-shifter unit. It has 32 general (16bit) registers and a 4kByte weight memory. There are 64PEs in one chip. 1, 8 or 16 bits can be used for weights and 8 or 16 bits for activation.

The performance of CNAPS on SOFM was reported by Hammerstrom and Nguyen in [15]. The figures are based on a 20MHz version of CNAPS. Best match using Euclidean distance measure, having N=512 nodes and M=256 elements per vector (δ=16), can be carried out in 215 µs. Making their CUPS figure comparable to others the performance will be about 183 MCUPS. A CNAPS computer can maximally achieve 10240 MIPS on dot-product operations. The efficiency is thus:

$$E = \frac{3.75\,(183)}{10240} = 7\%$$

More figures are needed to be able to analyse where the bottleneck is. The low efficiency may depend on the high maximal performance, achieved in an operation mode which can not be utilized for SOM.

5.2 Connection Machine

The Connection Machine [16, 50] manufactured by Thinking Machines Corporation (TMC) is for the moment the most massively parallel machine built (from 8k up to 64k processing elements). In addition to its large number of processors, two of its strong points are its powerful hypercube connection for general communication and the multidimensional mesh connection for problems with regular array communication demands. In the CM-2 model TMC also added floating point support, implemented as one floating point unit per 32 PEs. This means 2048 floating point units on a 64k machine, giving a peak performance of 10 GFlops. CM-2 is one of the most popular parallel computers for implementing ANN algorithms.

Obermayer et al. have implemented large SOM on the CM-2 [45]. They used node parallelism and up to 16k PEs (nodes). The input vector length M was varied and lengths of up to M=900 were tested. 48 MCUPS were achieved on a problem with n=N=16384 and M=100. As they used a bell-shaped Gaussian function for neighbourhood calculations, an efficiency measure according to equation (EQ 1) would be meaningless.

The same authors have also implemented the same algorithm on a self built computer with 60 T800 (Transputer) nodes connected in a systolic ring. Besides algorithmic analysis they have benchmarked the two architectures. Having M=100, n=30 and N=14400 they achieved 2.4 MCUPS.

The conclusion is that the CM-2 (16k PEs) with floating point support is equal to 510 Transputer nodes for the SOM. As a 16k CM-2 has 512 Weitek floating point units, each with approximately the same performance as one T800 on floating point calculations, it can be concluded that SOM basically is computation bound. In a "high-communication" variant of SOM where broadcast could not be used efficiently a 30 node Transputer machine would run at one third of the CM-2 speed.

5.3 L-Neuro

The Laboratoires d'Electronique Philips (LEP), Paris, have designed a VLSI chip called L-Neuro. It contains 16 processors working in SIMD fashion. In association with these chips Transputers are imagined as control and communication processors. The chip has support for multiplications with a multiply step. Weights are represented by 2-complement numbers over 8 or 16 bits, and the states of the neurons are coded over 1 to 8 bits.

Duranton and Sirat [7, 8] have described implementations of both SOM, Hopfield and BP networks on this architecture. However, no figures of performance were given.

5.4 MasPar

MasPar MP-1 [1, 5, 42] is a SIMD machine with both mesh and global interconnection style of communication. It has floating point support, both VAX and IEEE standards. The number of processing elements can vary between 1024 and 16384. Each PE has 40 32-bit registers, a 4-bit integer ALU, floating point "units" for Mantissa and Exponent, addressing unit for local address modifications, and a 4-bit broadcast bus. MP-1 has a peak performance, for a 16k PE machine, of 1500 MFlops single-precision [42].

In [4, 12, 13] Grajski, Chin et al. have implemented BP and SOM. The mapping of SOM into MP-1 uses node parallelism. It was measured to give 17.2 MCUPS on a 4k machine when 16-dimensional input vectors were used. They report that high efficiency is achieved using the MP-1 and that the performance figures increase with the dimensionality (up to 18 MCUPS).

5.5 REMAP^3

The goal of REMAP^3 is to construct modules for ANN computation. A typical module for SOM would consist of a few thousand bit-serial processors. If the processing elements needed for ANN were integrated on a VLSI chip more than 128 PEs per chip would be possible. On a single board, 1k processors with memory would be possible. If implemented on VLSI a clock frequency above 20 MHz would be of no problem. This makes it possible to achieve an update rate of approximately 539 per second on an $n = N = 2048$ problem (M=128 andδ = 16) and more than 11800 updates per second on the smaller problem ($n = N = 1024$, $M = 10$ andδ = 8). The corresponding CUPS (and efficiency) figures would be 141 MCUPS (=> E=83%) and 121 MCUPS (=> E=71%), respectively.

5.6 Transputer

The Transputer [22, 54] is a single chip 32-bit microprocessor. It has support for concurrent processing in hardware which closely corresponds to the Occam [2, 21, 23] programming model. It contains onchip RAM and four bi-directional 20 Mbits/sec communication links. By wiring these links together a number of topologies can be realized. Each Transputer of the T800 type is capable of 1.5 MFlops (20MHz) and architectures with up to 4000 Transputers are being built [54].

The SOM model has been implemented on Transputers by Hodges et al. [17], Siemon and Ultsch [48] and Obermayer et al. [45]. All three implementations distribute the nodes over the PEs and use ring communication to distribute the input vector and to find the global minimum. As long as the neighbourhood is larger than the number of PEs this mapping is quite efficient. Good performance will also be achieved for high input dimension.

Hodges et al. presented an equation for the performance but no concrete implementation.

Siemon and Ultsch state a performance of 2.7 MCUPS on a 16 Transputer machine. Having N=128x128, M=17, u=25000/2546 the total number of operations would be $O = 991000$. Each Transputer can give 1.5 MFlops so the maximum number of Flops would be 24MFlops. Then the efficiency would be E = 42%

Obermayer et.al have implemented SOM on, and compared the performance of, Connection Machine and Transputers, see section 5.2.

A more general implementation framework, called CARELIA, has been developed by Koikkalainen and Oja [35]. The neural network models are specified in a CSP-like formalism [33, 34, 35]. The simulator is currently running on a network of Transputers and one of the models implemented is SOM. The performance of the simulator has not been reported.

5.7 Warp

Warp is a one-dimensional array of 10 or more powerful processing elements (PEs) developed at Carnegie-Mellon University in 1984-87 [36]. Each cell/PE has a microprogrammable controller, a 5 MFlops floating-point multiplier, a 5 MFlops floating-point adder and a local memory. Communication between adjacent cells may be conducted in parallel over two independent channels: a left-to-right X channel and bidirectional Y channel.

An implementation of SOM on Warp has been described by R. Mann and Haykin [39]. When they used training example parallelism between 6 and 12.5 MCUPS were achieved. Because of a fixed communication overhead (0.01s) at the start of each batch, better performance could not be achieved. Some minor problem with the topology ordering process when using training example parallelism were reported. They suggested that either the map starts at some order instead of at random state, or that the map is trained sequentially for the first 100-1000 steps, after which the training example parallelism is “turned on”.

The Warp has 10 PEs, each having 10 MFlops, giving it a total of 100MFlops. Having N=1024 and M=128 the implementation could run at u=5.3x18 updates per second (each batch (epoch) was 18 training examples), giving it 12.5 MCUPS and an efficiency of E=47%.

5.8 TInMANN

Van den Bout et al. [41, 52, 53] have suggested a stochastic all digital implementation of Competitive Kohonen learning (see section 2.3) called TInMANN.

Their modifications to the SOFM model make it possible to use very simple PEs. A typical node would consist of two registers (10-12 bits), two adders or subtractors, two flags, memory for weights, gated broadcast, global-or function and some control logic. A VLSI version of TInMANN has been implemented where each node used 4000 transistors [41]. Using 10-bit weights the memory could be used for three to four weights per node, i.e. the input vector length M is very limited. The update rate, using 20MHz clock, is 267000 updates per second per node using $M = 3$. If one million transistors were used a 250 node chip could be constructed and thus giving 200 MCUPS per chip. As broadcast is used there is no problem to extend the system outside the chip boundaries.

A “rapid prototyping” version of the architecture in reconfigurable logic (XILINX) is also reported [53]. To eliminate some complexity and space, the architecture uses bit-serial nodes. Each chip (X3020) contains 3 PEs and external RAM is used for weights (i.e. much larger M can be used). One circuit board, called Anyboard, contains up to 8 X3020 and thus contains up to 21 PEs. This project was not completed. However, simple calculations showed that the interface between the host and the bit-serial nodes over the IBM PC bus was the major bottleneck. Ignoring the bus interface, the nodes were quite fast [51].

COMPUTER	n	Max MIPS (MFLOPS)	N	M	δ	u	MCUPS	E
CM	16384	2500	16384	100	FP		48	-%
MasPar	4096	376	4096	16	FP	250	16	16%
	4096	376	4096	256	FP	17	18	18%
Warp	10	100	1024	16	FP	522	8.6	32%
	10	100	1024	128	FP	95.4	12.5	47%
Transputer								
Obermayer	30	45	14400	100	FP		2.4	-%
Siemon	16	24	16384	17	FP	9.8	2.7	40%
CNAPS	256	10240	512	128	8		210	8%
	256	10240	512	128	16		180	7%
REMAP[3]	128	80	2048	128	8	67	17.5	82%
	1024	640	1024	10	8	11800	121	71%
	2048	1280	2048	128	8	1070	280	82%
	2048	640	2048	128	16	539	141	83%
TInMANN	250	-	250	3	10	267000	200	-%

Figure 3 The performance figures on SOFM simulations, for all the discussed computers. The efficiency *E* is calculated as 3.75 CUPS/IPS_{max}. No efficiency figure has been given if the CUPS figure is incompatible with the other figures.The number of processors is denoted by *n*, the number of nodes by *N*, the dimension of the input vector by *M*, the number of bits used for weights byδ and the update rate by *u*.
Note that if the computer manufacturer has been given “to high” maximum performance figures, the efficiency figure will be proportionally lower.

6.0 CONCLUSION

Unless especially tuned for SOM as REMAP[3], none of the computers studied have very high efficiency. As many of the performance figures given in the literature are measured under vastly different experimental setups, the figures given for MCUPS and efficiency, should be used cautiously. Still, the figures together with our analysis, indicate that the communication structure and the control mechanism are of little importance for the calculation of SOM. When designing a high performance SOM computer the design effort must be on implementing efficient (with respect to time and area) multipliers which can be supported with operands from the controller, e.g. using broadcast, together with high bandwidth access to local memory.

The fact that the neighbourhoodN_c becomes very small towards the end of the training session seems easier to take advantage of on a coarse grain MIMD computer. Even if only one PE is working there will be relatively fewer idle processors during the updating step, thus giving the MIMD computer a possibly better efficiency.

The analysis also indicates that SOM can be mapped efficiently onto bit-serial SIMD computers. The only requirement is that a bit-serial multiplier is included to support the many multiplications.

By modifying the algorithm to include a “stochastic signal” it is possible to run competitive learning without using multiplication. This makes it possible to achieve enormous update rates, but as the updates are of a different kind, it is difficult to compare it to the original method with respect to speed. Still, this modification is very interesting as an alternative to ordinary competitive learning algorithms, as it reduces the architectural components needed for the computation. Moreover, bit-serial SIMD computers could be considered as one of the prime candidates for an efficient implementation of this modified SOM algorithm.

The natural and totally dominating dimension of parallelism is the node parallelism. Training example parallelism has been used, but it seems that only relatively small batch sizes may be used as there will otherwise be failures in the topological ordering. The weight parallelism and the mixed mapping, which were successfully used in the computation of Kanerva’s SDM model [43], can not be used efficiently for the calculation of the SOM model. This is due to the fact that the adaptation is taking place in the weight matrix which is also used for the selection phase, whereas for SDM the adaptation is made in a separate matrix.

7.0 REFERENCES

[1] Blank, T. "The MasPar MP-1 Architecture." In *Proceedings of COMPCON Spring 90*, pp. 20-24, San Francisco, CA, 1990.

[2] Bowler, K. C., et al. *An Introduction to OCCAM 2 Programming*. Chartwell-Bratt. 1987.

[3] Bradburn, D. S. "Reducing transmission error effects using a self-organizing network." In *International Joint Conference on Neural Networks*, Vol. 2, pp. 531-537, Washington, DC, 1989.

[4] Chinn, G., et al. "Systolic array implementations of neural nets on the maspar MP-1 massively parallel processor." In *International Joint Conference on Neural Networks*, Vol. 2, pp. 169-173, San Diego, 1990.

[5] Christy, P. "Software to support massively parallel computing on the MasPar MP-1." In *Proceedings of COMPCON Spring 90*, pp. 29-33, San Fransisco, CA, 1990.

[6] DeSieno, D. "Adding a conscience to competitive learning." In *International Conference on Neural Networks*, Vol. 1, pp. 117-124, San Diego, 1988.

[7] Duranton, M. and J. A. Sirat. "Learning on VLSI: A general purpose digital neurochip." In *International Conference on Neural Networks*, Washingtin, DC, 1989.

[8] Duranton, M. and J. A. Sirat. "Learning on VLSI: A gerneral-purpose digital neurochip." *Philips Journal of Research*. Vol. 45(1): pp. 1-17, 1990.

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

[10] Flynn, M. J. "Some computer organizations and their effectiveness." *IEEE Transaction on Computers*. Vol. C-21: pp. 948-60, 1972.

[11] Grajski, K. A. "Neurocomputing using the MasPar MP-1." (Technical Report No. 90-010), Ford Aerospace, 1990.

[12] Grajski, K. A. "Neurocomputing using the MasPar MP-1." *Digital Parallel Implementations of Neural Networks.* Przytula and Prasanna ed. Prentice-Hall. (Forthcoming). 1992.

[13] Grajski, K. A., et al. "Neural Network Simulation on the MasPar MP-1 Massively Parallel Processor." In *The International Neural Network Conference*, Paris, France, 1990.

[14] Hammerstrom, D. "A VLSI architecture for high-performance, low-cost, on-chip learning." In *International joint conference on neural networks*, Vol. 2, pp. 537-543, San Diego, 1990.

[15] Hammerstrom, D. and N. Nguyen. "An implementation of Kohonen's self-organizing map on the Adaptive Solutions neurocomputer." In *International Conference on Artificial Neural Networks*, Vol. 1, pp. 715-720, Helsinki, Finland, 1991.

[16] Hillis, W. D. and G. L. J. Steel. "Data parallel algorithms." *Communications of the ACM*. Vol. 29(12): pp. 1170-1183, 1986.

[17] Hodges, R. E., C.-H. Wu and C.-J. Wang. "Parallelizing the self-organizing feature map on multi-processor systems." In *International Joint Conference on Neural Networks*, Vol. 2, pp. 141-144, Washington, DC, 1990.

[18] Hopfield, J. J. "Neural networks and physical systems with emergent collective computational abilities". *Proceedings of the National Academy of Science USA*. 79: pp. 2554-2558, 1982.

[19] Hopfield, J. J. "Neurons with graded response have collective computational properties like those of two-state neurons". *Proceedings of the National Academy of Science USA*. 81: pp. 3088-3092, 1984.

[20] Hopfield, J. J. and D. Tank. "Computing with neural circuits: A model." *Science*. Vol. 233: pp. 624-633, 1986.

[21] INMOS Limited. *Occam programming model*. Prentice-Hall. 1984.

[22] INMOS Limited. "The Trasputer family 1987". 1987.

[23] INMOS Limited. *Occam 2 Reference Manual*. Prentice-Hall. London. 1988.

[24] Johnson, M. J., N. M. Allinson and K. J. Moon. "Digital realisation of self-organising maps." In *Neural Information Processing Systems 1*, pp. 728-738, Denver, CO, 1988.

[25] Kanerva, P. *Sparse Distributed Memory.* MIT press. Cambridge, MA. 1988.

[26] Kangas, J. A., T. K. Kohonen and J. T. Laaksonen. "Variants of self-organizing maps." *IEEE Transaction on Neural Networks*. Vol. 1(1): pp. 93-99, 1990.

[27] Kohonen, T. "The 'neural' phonetic typewriter." *Computer*. Vol. 21(3): pp. 11-22, 1988.

[28] Kohonen, T. *Self-Organization and Associative Memory.* (2nd ed.) Springer-Verlag. Berlin. 1988.

[29] Kohonen, T. "Improved versions of learning vector quantization." In *International Joint Conference on Neural Networks*, Vol. 1, pp. 545-550, San Diego, 1990.

[30] Kohonen, T. "The self-organizing map." *Proceedings of the IEEE*. Vol. 78(9): pp. 1464-1480, 1990.

[31] Kohonen, T. "Some practical aspects of the self-organizing maps." In *International Joint Conference on Neural Networks*, Vol. 2, pp. 253-256, Washington, DC, 1990.

[32] Kohonen, T., et al. "An adaptive discrete-signal detector based on self-organizing maps." In *International Joint Conference on Neural Networks*, Vol. 2, pp. 249-252, Washington, DC, 1990.

[33] Koikkalainen, P. "MIND: a specification formalism for neural networks." In *International Conference on Artificial Neural Networks*, Vol. 1, pp. 579-584, Helsinki, Finland, 1991.

[34] Koikkalainen, P. and E. Oja. "Specification and implementation environment for neural networks using communication sequential processes." In *International Conference on Neural Networks*, San Diego, CA, 1988.

[35] Koikkalainen, P. and E. Oja. "The CARELIA simulator: a development and specification environment for neural networks." (Research Report No. 15/1989), Lappeenranta Univ. of Tech, Finland, 1989.

[36] Kung, H. T. "The Warp computer: architecture,implementation and performace." *IEEE Transaction on Computers*. Vol. Dec: 1987.

[37] Mann, J. "The effects of circuit integration on a feature map vector quantizer." In *Neural Information Processing Systems 2*, pp. 226-231, Denver, CO, 1989.

[38] Mann, J. and S. Gilbert. "An analog self-organizing neural network chip." In *Neural Information Processing Systems 1*, pp. 739-747, Denver, CO, 1988.

[39] Mann, R. and S. Haykin. "A parallel implementation of Kohonen feature maps on the Warp systolic Computer." In *International Joint Conference on Neural Networks*, Vol. 2, pp. 84-87, Washington, DC, 1990.

[40] Martinetz, T. M., H. J. Ritter and K. J. Schulten. "Three-dimensional neural net for learning visuomotor coordination of robot arm." *Transaction on neural networks*. Vol. 1(1): pp. 131-136, 1990.

[41] Melton, M., et al. "VLSI Implementation of TInMANN." In *Advances in Neural Information Processing Systems 3*, Denver, CO, 1990.

[42] Nickolls, J. R. "The design of the MasPar MP-1: a cost effective massively parallel computer." In *Proceedings of COMPCON Spring 90*, pp. 25-28, San Fransisco, CA, 1990.

[43] Nordström, T. "Sparse distributed memory simulation on REMAP3." (Research Report No. TULEA 1991:16), Luleå University of Technology, Sweden, 1991.

[44] Nordström, T. and B. Svensson. "Using and designing massively parallel computers for artificial neural networks." (Research Report No. TULEA 1991:13), Luleå University of Technology, Sweden, 1991.

[45] Obermayer, K., H. Ritter and K. Schulten. "Large-scale simulations of self-organizing neural networks on parallel computers: application to biological modelling." *Parallel Computing*. Vol. 14(3): pp. 381-404, 1990.

[46] Ritter, H. J., T. M. Martinetz and K. J. Schulten. "Topology conserving maps for learning visuo-motor-coordination." *Neural Networks*. Vol. 2(3): pp. 159-168, 1989.

[47] Rumelhart, D. E. and J. L. McClelland. *Parallel Distributed Processing; Explorations in the Microstructure of Cognition.* Vol I and II MIT Press. Cambridge. 1986.

[48] Siemon, H. P. and A. Ultsch. "Kohonen networks on transputers: Implementation and animation." In *International Neural Network Conference*, Vol. 2, pp. 643-646, Paris, 1990.

[49] Svensson, B. and T. Nordström. "Execution of neural network algorithms on an array of bit-serial processors." In *10th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*, Vol. II, pp. 501-505, Atlantic City, New Jersey, USA, 1990.

[50] Thinking Machines Corporation. "Connection Machine, Model CM-2 Technical Summary." (Version 5.1), T M C Cambridge, Massachusetts, 1989.

[51] Van den Bout, D. E. 1991. Personal communication.

[52] Van den Bout, D. E. and T. K. M. III. "TInMANN: The integer Markovian artificial neural network." In *International joint conference on neural networks*, Vol. 2, pp. 205-211, Washington, 1989.

[53] Van den Bout, D. E., W. Snyder and T. K. Miller III. "Rapid prototyping for neural networks." *Advanced Neural Computers*. Eckmiller ed. North-Holland. Amsterdam. 1990.

[54] Whitby-Stevens, C. "Transputers — past, present, and future." *IEEE Micro*. (December): pp. 16-82, 1990.

[55] Wilson, S. S. "Neural computing on a one dimesional SIMD array." In *11:th International Joint Conference on Artificial Intelligence*, pp. 206-211, Detroit, Michigan, USA, 1989.

Svensson, B. and T. Nordström, "Execution of neural network algorithms on an array of bit-serial processors," in 10*th International Conference on Pattern Recognition, Computer Architectures for Vision and Pattern Recognition*, Atlantic City, NJ, USA, 1990, vol. II, pp. 501-505, ISBN 0-8186-2063-3.

EXECUTION OF NEURAL NETWORK ALGORITHMS ON AN ARRAY OF BIT-SERIAL P ROCESSORS

B. Svensson T. Nordström

Division of Computer Engineering
Department of Systems Engineering and Mathematics
Luleå University of Technology, Sweden

E-mail: bertil@sm.luth.se or tono@sm.luth.se

ABSTRACT

Large processor arrays are candidates for performing computations of neural network models at speeds required for real time applications, e.g. in pattern recognition. The paper gives a general model of an array of bit-serial processors and demonstrates the mapping of neural net models on such an array.

The approach maps a neuron on each processing element and makes communication all-to-all available by connection weight matrices. The required communication structure is very simple.

The bit-serial approach allows trade-offs between speed and precision, even dynamically. Performance figures are given. A bit-serial multiplier is an important part of the design. Implementation aspects are discussed and it is shown that a one-board realization of a 1024 processor system is feasible with current, commonly available, technology.

INTRODUCTION

Recent years have seen an enormous increase of interest in neural networks. It has been realized that massive parallelism is required for human-like performance in pattern recognition. Neural networks provide one technique to do this. Processor arrays are candidates for performing the computations efficiently. The subject of this paper is to study the mapping of neural network computations on a regular array of a large number of simple processors. The computations are uniform and arithmetically simple. This suggests that simple processing elements are sufficient and that the SIMD type of architecture is appropriate. The number of interconnections in a neural network is often orders of magnitude greater than the number of processing units. This suggests that connectivity be stored partly in matrices.

We study the mapping of both feedforward (with back-propagation) and feedback neural nets. The characteristics of these models will be briefly outlined. Before that we introduce a generic architecture for a bit-serial array processor (BAP). We describe the algorithms in a parallel language (Pascal/L) which includes constructs directly implementable as elementary operations of a BAP. The computations are analysed, performance figures are given, and system implementation is discussed.

BIT-SERIAL ARRAY PROCESSORS

A Bit-serial Array Processor (BAP) is characterized by the following properties:

- It is organized as an SIMD processor, i.e. it consists of many processing elements (PEs) and one common control unit.
- The PEs treat data bit-serially and the data paths to and from each PE are only one bit wide.
- Activation of the PEs may be data driven ("associative process"), which means it is not the location or address of a PE that decides its action on an instruction from the control unit, but some property of the data in the memory or registers of that PE.
- An interconnection network defines a topological relationship between PEs.

A commonly used organization of the BAP is to place the interconnection network between the memory part and the logic part of the PE as shown in Figure 1. This does not mean that memory and logic are physically apart - on the contrary, they are seen as a whole and should preferably be put on the same chip.

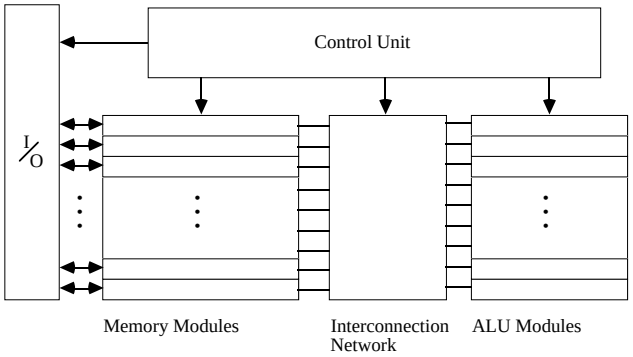

Figure 1. Organization of a Bit-serial Array Processor.

A BAP is defined by the characteristics of five parts: data storage, processing, data alignment, input/output, and control.

Data storage is organized as Memory Modules (MMs). One bit from each MM is accessible at a time. A number of such *bit-slices*, normally consecutive, form a *field*.

The *Processing part* is an ensemble of Arithmetic and Logic Units (ALUs) which implement functions on a set of one-bit arguments. The complexity of the ALU may vary from boolean functions of two variables through bit-serial multipliers to full bit-serial floating point units.

One of the registers in the ALU is the one-bit *Activity Register*, the contents of which determines whether or not the ALU takes part in the specified operation. To choose only one ALU for activity, a *select first* facility is included. In more elaborate models multi-bit registers may be used to determine one out of a set of actions to be performed.

The *Data Alignment part* consists of an interconnection structure that allows each ALU to receive data also from "neighbouring" modules. Common structures are the square grid, the linear array, the n-cube and the shuffle-exchange. Many separate structures may be implemented on the same processor.

The structure of the *Input/Output part* design is strongly dependent on the demands of the application and may be varied in several ways. For example, in some cases a direct bit-slice wide interface to the data source may be motivated.

The total activity is mastered by the *Control Unit*, which takes instructions from an ordinary sequential processor. The most obvious task for the Control Unit is to translate operations on data items (e.g. vectors and matrices) to sequences of bit operations. This should be performed without any overhead.

Based on the types of operands and results six basic types of instructions to manipulate data in the array can be identified:

Instruction type	*Example*
field —> field	Increment field, Permute field
field —> selector	Max/min value of a field
field,field —> field	Multiply fields, Pairwise max
field,field —> selector	Pairwise equality
constant,field —> field	Multiply by constant
constant,field —> selector	Closest match, Greater than

Multiplication is a frequent operation in many application areas. Using ALUs with a complexity comparable to a full adder only, the multiplication time grows quadratically with the data length. Ohlsson [1] suggested a bit-serial multiplier in each ALU, giving a multiplication time that is no longer than the time required to read the operands and store the result.

Figure 2 shows the design for multiplication of two 2's complement integers using a series of full adders (FA). The multiplicand is first shifted in, most significant bit first, into the array of M flip-flops. The multiplier is then applied to the input, least significant bit first, and the product bits appear at the output, least significant bit first. The S flip-flops store the accumulated sum. A more detailed description is given in [1] and [2].

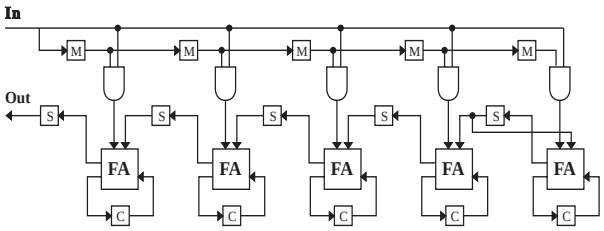

Figure 2. A 5 bit wide bit-serial multiplier using carry-save technique. M, S and C are flip-flops and FA are full adders.

NEURAL NETWORK ALGORITHMS

Several neural net models have been proposed. They are characterized by network topology, node characteristics, and training rules. Frequently used and discussed models are the *multilayer feedforward networks* with supervised learning by error back-propagation [3] and the *feedback networks* , either with symmetric connectivity and stochastic nodes (Boltzmann machines [4, 5]), symmetric connectivity and deterministic nodes (Hopfield net [6, 7, 8]), or nonsymmetric connectivity and deterministic nodes[9, 10].

In order to be as general as possible in the implementation studies we use a feedback algorithm without any assumption on symmetry of the weight matrix. Thus, for the Hopfield model and the Boltzmann machine shorter execution times than those reported below can be expected (both use symmetric matrices).

The back-propagation model is used as a pattern classifier or feature detector. The feedback models are used as auto-associative memories for tasks like pattern completion.

Feedforward networks with error back-propagation

A feedforward net (ff net) with four layers is shown in Figure 3. Each node (neuron) in a layer receives input from every node in the previous layer. Each node computes a weighted sum of all its inputs. Then it applies a nonlinear activation function to the sum, resulting in an activation value of the neuron. A sigmoid function, with a smooth threshold like curve, is the most frequently used activation function in feedforward networks.

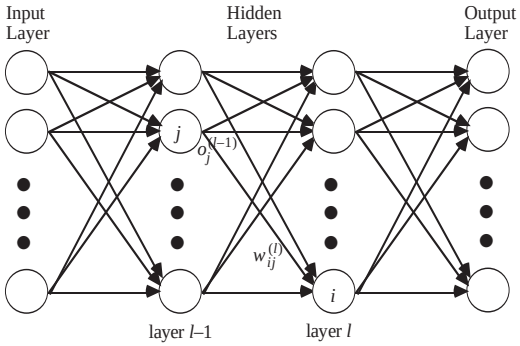

Figure 3. A four-layer feedforward network.

The back-propagation algorithm (also known as the generalized delta rule) [3] is used to train the network in our examples.

In the first phase the input to the network is provided and values propagate through the network to compute the output vector *O*. *O* is then compared with a target vector *T* provided by a teacher, resulting in an error vector *E*. In the second phase the values of the error vector are propagated back. The error signals for hidden units are thereby determined recursively: Error values for layer l are determined from a weighted sum of the values of layer l+1, again using the connection weights – now "backwards". The weighted sum is multiplied by the derivative of the activation function to give the error value δ.

Finally, appropriate changes of weights and thresholds are made. The weight change in the connection to unit i in layer l from unit j in layer l–1 is proportional to the product of the output value o and the error value δ: $\Delta w_{ij}^{(l)} = \eta \delta_i^{(l)} o_j^{(l-1)}$. The bias (threshold) value may be seen as the weight from a unit that is always on. The algorithm is summarized below.

1. Apply input
2. Compute output $o_j^{(l)} = f\left(net_j^{(l)} + b_j^{(l)}\right)$

   where $net_j^{(l)} = \sum_i w_{ij}^{(l)} o_i^{(l-1)}$ for each layer.
3. Determine error vector $E = T - O$
4. Propagate error backwards.

   If node j is an output node then the elements of the error value vector D are

   $\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)})(t_j^{(l)} - o_j^{(l)}) = o_j^{(l)}(1 - o_j^{(l)})e_j^{(l)}$

   else

   $\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)})\sum_i \delta_i^{(l+1)} w_{ij}^{(l+1)}$

   Here we have used the fact that the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ has the derivative $f' = f(1-f)$
5. Adjust weights and thresholds.

   $\Delta w_{ij}^{(l)} = \eta \delta_i^{(l)} o_j^{(l-1)}$, $\Delta b_i^{(l)} = \eta \delta_i^{(l)}$
6. Repeat from 1.

Algorithm 1. Back-propagation training algorithm

Feedback networks

A feedback network has a single set of completely interconnected nodes, see Figure 4. All nodes are both input and output nodes. Each node computes a weighted sum of all its inputs and applies a nonlinear activation function to the sum. The resulting value is treated as input to the network in the next step. When the net has converged, i.e. when the output no longer changes, the pattern on the output of the nodes is the network response.

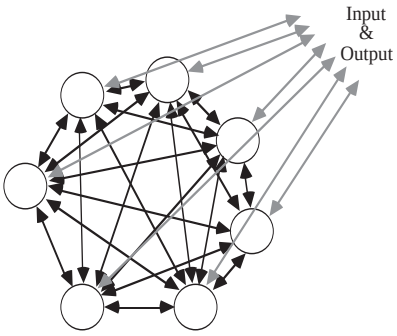

Figure 4. A seven node feedback network.

Training or learning can be done in supervised mode with the delta rule [11] or back propagation [10], or unsupervised by a Hebbian rule [11]. The delta rule is more powerful than the Hebb rule, and more commonly used than back-propagation (for feedback nets). We only analyse the delta rule algorithm.

In the first phase of training the pattern is imposed on the net at time zero by forcing the output from the net to match the pattern. Following this initiation, the net iterates in discrete time steps using the given formula. When the net has converged the activation a_i is compared to the target t_i and the error is calculated as $e_i = t_i - a_i$. The weights are changed in proportion to the product of the activation a_j and the error e_i, i.e. $\Delta w_{ij} = \eta e_i a_j$

1. Set activation values to external input values
2. Calculate new activation values $a_j = f(net_j + b_j)$ where $net_j = \sum_i w_{ij} a_i$, until the network is stable
3. Determine error vector $E = T - A$
4. Adjust the weights $\Delta w_{ij} = \eta e_i a_j$ and bias $\Delta b_i = \eta e_i$.
5. Repeat from 1.

Algorithm 2. Delta learning algorithm for feedback networks

MAPPING NEURAL NETWORKS ON A BAP

It should be clear from the above descriptions of networks, that the computations of both a feedforward network with error back-propagation and a feedback network involve mainly matrix-by-vector multiplications, where the matrices contain the connection weights and the vectors contain activation values or error values. Such a multiplication contains N^2 scalar multiplications and N computations of sums of N numbers.

The fastest possible way to compute this is to perform all N^2 multiplications in parallel, which requires N^2 PEs and unit time, and then form the sums by using trees of adders. The addition requires $N(N\text{-}1)$ adders and $\boldsymbol{O}(\log N)$ time. This is, however, an unrealistic method depending on both the number of PEs required and the communication problems caused. Instead we take the approach of having as many PEs as neurons in a layer, N, and storing the connection weights in matrices, sized N by N, one for each layer. The PE with index j has access to row j of the matrix by accessing its own memory word.

Feedforward net with error back-propagation

Figure 5 shows the data storage for each layer. In the forward pass the *net* vector is computed by N successive, parallel multiply-and-add operations, each requiring access to a different output value from the previous layer. Thus the PEs must, one after the other, broadcast their output values to all PEs. The O-vector is computed by a parallel application of the activation function.

$$\begin{array}{cccc} \mathbf{W}^{(l)} & & B^{(l)} \; O^{(l)} \; net^{(l)} \; E^{(l)} & \\ w_{00}^{(l)} \; w_{01}^{(l)} \cdots w_{0,N-1}^{(l)} & & b_0^{(l)} \; o_0^{(l)} \; net_0^{(l)} \; e_0^{(l)} & \Longleftrightarrow \mathrm{PE}_0 \\ w_{10}^{(l)} \; w_{11}^{(l)} \quad\; w_{1,N-1}^{(l)} & & b_1^{(l)} \; o_1^{(l)} \; net_1^{(l)} \; e_1^{(l)} & \Longleftrightarrow \mathrm{PE}_1 \\ \vdots \qquad \ddots \quad \vdots & & \vdots \quad \vdots \quad \vdots \quad \vdots & \vdots \\ w_{N-1,0}^{(l)} \; w_{N-1,1}^{(l)} \cdots w_{N-1,N-1}^{(l)} & & b_{N-1}^{(l)} \; o_{N-1}^{(l)} \; net_{N-1}^{(l)} \; e_{N-1}^{(l)} & \Longleftrightarrow \mathrm{PE}_{N-1} \end{array}$$

Figure 5. Data storage for layer l.

In the backward pass the computation of the error vector for each layer requires vertical addition. We suggest a bit-serial adder tree. The addition of each column can be overlapped with the multiplications for the next. On completion of this phase, weight changes are calculated. The O-vector of layer l–1 is first multiplied by a constant, η. Then, for each j, the j:th value of the result is broadcast to all other PEs, the E-vector is multiplied by this value and the result is added to the j:th column of the weight matrix. The threshold vector is changed in a similar way.

The Pascal/L language

Pascal/L is an extension of Pascal for parallel processing, developed in the LUCAS project [2]. In Pascal/L the parallelism of the architecture has a correspondence in the syntax of the language. Thus, constructs in the language are directly implementable as elementary operations of a BAP.

A **selector** defines a boolean vector over the MMs and is used to control the parallelism of operations. A **parallel array** has a fixed number of components, all of the same type and located in the MMs. An indexing scheme allows simultaneous access to a column or a subset of the column components of a two-dimensional array. For example: **W**[*,5] selects column 5 of **W**, **W**[SEL,5] selects a subset of column 5. A parallel array may be used without any index at all (and no brackets), in which case all components of the array are referenced.

To support data-driven processing a number of standard functions and procedures can be applied to selectors. The **first** function finds the first component of a selector with the value *true* and returns a new selector with only this element true. The **next**-procedure assigns *false* to the first true element of the selector. This is useful when elements are to be processed sequentially. The **some** function returns *true* if there is at least one true element of the selector, otherwise it returns *false*.

The Pascal/L program

For the feedforward net, the following declarations are needed (For simplicity of notation we consider only one layer):

```
var  W  : parallel array [0..N–1,0..N–1] of integer(b);
     net : parallel array [0..N–1] of integer(b + logN);
     B,O,O⁻,E,E⁺:parallel array[0..N–1] of integer(b);
{ O⁻ is the output of the previous layer, E⁺ is the error of the
following layer }
     sel : selector [0..N–1];
     j   : integer;
```

The following program parts implement the algorithm:

```
Forward pass:
      j := 0; net := 0; sel := TRUE;
      while some(sel) do begin
          net := net + W[*,j] * O-[first(sel)]; next(sel); j := j+1;
      end;
      O := f(net+B);
Backward pass:
 Computation of error vectors:
      j := 0; E := 0; sel := TRUE;
      while some(sel) do begin
          E:= E + E+[first(sel)] * W[*,j]; next(sel); j:=j+1;
      end;
      E := O*(1–O)*E;
Computation of new weights:
      j := 0; sel := TRUE;
      while some(sel) do begin
          W[*,j] := W[*,j] + η*O-[first(sel)]*E; next(sel); j:=j+1;
      end;
      B := B + η*E;
```

General feedback algorithm

The data storage required for a general feedback network is approximately the same as for one layer of the ff network. The computations of the forward pass are the same as in the ff net (one layer), but are repeated until no more changes occur, or for a fixed number of times. To update the weights, the error vector is first calculated in parallel and then multiplied by η, also in parallel. The values of the activation vector are then broadcast one by one to all other PEs. Each PE multiplies the value with its (η*error)-value and adds the result to the corresponding weight.

The Pascal/L program

```
var W   : parallel array [0..N-1,0..N-1] of integer(b);
    net : parallel array [0..N-1] of integer(b + logN);
    B,A,Aold,E : parallel array [0..N-1] of integer(b);
    ExtInputs  : parallel array [0..N-1] of integer(b);
    sel : selector [0..N-1];
    j,m : integer;
{Initialize}
    rand(W);            {Initialize the weights}
    A := ExtInputs;     {Set activation values}
    Aold := A+2*e;      {Get past first while test}
{Train the net one pass with one pattern
Calculate new activation values during m cycles or to convergence}
    m := 0;
    while some(abs(Aold - A) > e) and (m < mmax) do begin
        j:= 0; net := 0; sel := TRUE; Aold := A;
        while some(sel) do begin
          net := net + W[*,j]*A[first(sel)]; next(sel); j := j+1;
        end;
        A := f(net+B); m := m+1;
    end;
{Update weights}
    E := η*(ExtInputs - A); j := 0; sel := TRUE;
    while some(sel) do begin
        W[*,j] := W[*,j] + A[first(sel)]*E; next(sel); j:=j+1;
    end;
    B:= B + η*E;
```

COMPUTATION TIME

Feedforward networks with error back-propagation

The computations for one layer of the feedforward pass contain N operations of type multiply(by constant)-and-add followed by a few (maximum ten) multiply-and-add operations to compute the activity function (e.g. by a piecewise linear activation function which approximates the sigmoid function). Since N is large in the applications we consider we can leave the latter operations out when we estimate the computation time.

During accumulation the sum will grow to a maximal length of b+log$N$ bits. On the average the number of cycles for multiply-and-add will be $4b$+log$N$–1, using the bit-serial multiplier of Figure 2.

In the backward error computation phase the summation is made over the adder tree in b+log$N$ cycles. A multiplication and a tree addition can be made simultaneously. The weight changing phase, finally, takes $4b$ cycles per column.

In total the computations for one layer consume: $[8b + \log N - 1 + \max(3b, b+\log N)]N$ cycles during training and $(4b + \log N - 1)N$ cycles during recall.

Assuming a clock frequency of 10 MHz (which is fairly conservative) execution times shown in Table 1 are derived.

A common measure of the performance of neural net hardware is the number of "CPS" (Connections per second). In a net with N neurons per layer, N^2 connections are used and/or updated in each layer. The MegaCPS figures for BAPs of different sizes are given in Table 2.

		N		
		256	1024	4096
b	8	2.4	9.9	40.6
	12	3.6	14.4	58.6
	16	4.7	18.9	76.6

a)

		N		
		256	1024	4096
b	8	1.0	4.2	17.6
	12	1.4	5.8	24.2
	16	1.8	7.5	30.7

b)

Table 1. a) Training time per layer (ms). b) Recall time per layer (ms). 10 MHz clock frequency is assumed.

		N		
		256	1024	4096
b	8	27	106	413
	12	18	73	286
	16	14	55	219

Training

		N		
		256	1024	4096
b	8	66	250	953
	12	47	180	694
	16	36	140	546

Recall

Table 2. Number of MegaCPS (Million Connections Per Second) for different network sizes and data precisions.

Feedback networks with delta rule.

The computations of one iteration contain N multiply-and-add operations. We do these computations in $(4b+\log N-1)N$ cycles. Thus, the figures of Table 1b apply. The weight changing phase takes $4bN$ cycles. Executing m iterations and one weight update then takes $m(4b + logN–1)N + 4bN$ cycles.

Table 3 shows the training times for different network sizes and precisions. The times for recall equal those of Table 1b.

MEMORY REQUIREMENTS AND IMPLEMENTATION ASPECTS

The amount of storage per word required for the feedback network, or for each layer in the ff net, is approximately bN. See Table 4. An m layer network requires m–1 times as much memory

		N		
		256	1024	4096
b	8	1.8	7.4	30.7
	12	2.6	10.8	43.8
	16	3.5	14.0	56.9

a)

		N		
		256	1024	4096
b	8	101	423	1770
	12	142	586	2440
	16	183	754	3100

b)

Table 3. Training times (ms) for feedback network. a) with 1 iteration, b) with 100 iterations. 10 MHz clock frequency is assumed.

(with a m–1 connection weight matrices stored). It should also be mentioned that there are methods proposed that include a *momentum* term α in the weight changing rule for the ff networks (refer to Algorithm 1):

$$\Delta w_{ij}^{(l)}(t) = \eta \delta_i^{(l)} o_j^{(l-1)} + \alpha \, \Delta w_{ij}^{(l)}(t\text{-}1)$$

Thus the past weight change affects the current direction to an amount determined by the constant α. This is considered to allow high learning rate without leading to oscillations. However, it requires that the weight changes be stored as well, which doubles the required memory space.

		N		
		256	1024	4096
b	8	2	8	32
	12	3	12	48
	16	4	16	64

Table 4. Memory requirements (kilobits) per processor for one layer (approximate figures).

Using commercially available RAM chips for the large amount of memory needed offers obvious advantages. On the other hand, memory on the same chip as the PEs can increase processing speed significantly.

If external RAM is used 64 PEs of the complexity we discuss can easily be integrated on one VLSI chip. A 1024 PE array will have 16 such chips, each with approximately 100 pins. Memory can be implemented using chips with 64k x 4 bits, giving a chip count of 256. With appropriate mounting technology such a network may be implemented on one board. It would run a four-layered feedforward network with 1024 neurons per layer at the speed of 34 training examples or 80 recall examples per second.

An implementation of a prototype BAP is currently being made as a joint project between Luleå University of Technology and Centre for Computer Science at Halmstad University College [12]. The implementation is software configurable to allow for "compilation" of a certain architecture to suit a specific application. Neural network computations constitute one such application area.

CONCLUSIONS

We have given a general model of a bit-serial array processor (BAP) and have shown how the computations of different neural network models can be performed on such a processor. A major advantage of the bit-serial working mode is that precision can be traded for speed. We have calculated execution times and memory requirements for feedforward and feedback networks of different sizes and with different numerical precision. Results show a large speed advantage over commercial neural net simulators and form the basis for the outline of a one-board implementation comprising 1024 processing elements.

An interesting result is that the computations do not require the processor 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 (a select first chain), and a bit-serial adder tree to add the values of a field.

The approach taken is to map one neuron on each processor — in the case of multilayer networks the same processors are used for all layers. If more processors are available, or if the processors are fewer than the neurons, the programs presented must be slightly changed. The speed will increase or degrade accordingly. Thus, the speed of a certain network can be adjusted by choosing the number of processors.

A critical operation in the computations is multiplication. We have shown how a very simple bit-serial multiplier structure using carry-save technique can equalize multiplication time relative to addition time.

It is seen that the different net models that we have studied put the same demands on the processing array. These models are representative for the neural networksarea, implying that efficient execution of most kinds of neural networks on a BAP can be expected.

Acknowledgment: Part of this research is financed by Halmstad University College and STU under contract no. 88-03901P

REFERENCES

[1] Ohlsson, L. "An improved LUCAS architecture for signal processing." (Technical report), Dept. of Comp.Eng., Univ. of Lund, 1984.

[2] Fernström, C., I. Kruzela and B. Svensson. *LUCAS Assosiative Array Processor - Design, Programming and Application Studies*. Vol 216 of *Lecture Notes in Computer Science*. Springer Verlag. Berlin. 1986.

[3] Rumelhart, D. E. and J. L. McClelland. *Parallel Distributed Processing; Explorations in the Microstructure of Cognition*. Vol I and II, MIT Press. Cambridge. 1986.

[4] Hinton, G. E. and T. J. Sejnowski. "Optimal perceptual inference." In *Proceedings of the IEEE Computer Soci ety Conference on Computer Vision & Pattern Recognition*, pp. 448-453, Washington, D.C., 1983.

[5] Hinton, G. E. and T. J. Sejnowski. "Learning and relearning in Boltzmann machines." In vol. II of [3].

[6] Hopfield, J. J. "Neural networks and physical systems with emergent collective computational abilities". *Proceedings of the National Academy of Science USA*. 79: pp. 2554-2558, 1982.

[7] Hopfield, J. J. "Neurons with graded response have collective computational properties like those of two-state neurons". *Proceedings of the National Academy of Science USA*. 81: pp. 3088-3092, 1984.

[8] Hopfield, J. J. and D. Tank. "Computing with neural circuits: A model." *Science*. Vol. 233: pp. 624-633, 1986.

[9] Pineda, F. J. "Generalization of Back-Propagation to Recurrent Neural Networks." *Phys. Review Letters*. Vol. 59(19): pp. 2229-2232, 1987.

[10] Almeida, L. D. "Backpropagation in perceptrons with feedback." In *NATO ASI series: Neural Computers*, Neuss, F.R. Germany, 1987.

[11] Rumelhart, D. E. and J. L. McClelland. *Explorations in Parallel Distributed Processing*. MIT Press. Cambridge, Massachusetts. 1988

[12] Svensson, B. "Implementation and application of a software configurable massively parallel computer." *Second Swedish Workshop on Computer Systems Architecture*, Bålsta, Sweden, 1989.