Exploiting Domain-Specific Properties: Compiling Parallel Dynamic Neural Network Algorithms into Efficient Code

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: -7343

March 17, 1999

Final accepted version of IEEE TPDS paper 105491

Submitted 1996-01-12, revised 1997-08-12 and 1998-08-17, accepted 1999-03-16

Abstract

Domain-specific constraints can be exploited to implement compiler optimizations that are not otherwise feasible: Compilers for neural network learning algorithms can achieve near-optimal co-locality of data and processes *and* near-optimal balancing of load over processors, even for dynamically irregular problems. This is impossible for general programs, but restricting programs to the neural algorithm domain allows for the exploitation of domain-specific properties: The operations performed by neural algorithms are broadcasts, reductions, and object-local operations only; the load distribution is regular with respect to the (perhaps irregular) network topology; changes of network topology occur only from time to time.

A language, compilation techniques, and a compiler implementation on the MasPar MP-1 are described; quantitative results for the effects of various optimizations used in the compiler are shown. Conservative experiments with weight pruning algorithms yield performance improvements of 27% due to load balancing; 195% improvement is achieved due to data locality, both compared to unoptimized versions. Two other optimizations, connection allocation and selecting the number of replicates, speed programs up by about 50% or 100%, respectively.

This work can be viewed as a case study in exploiting domain-specific information; some of the principles presented here may apply to other domains as well.

Keywords: compiler optimizations, high-level parallel language, irregular problems, dynamic data structures, communication optimization.

1 Introduction

The field of neural networks could in principle benefit a lot from parallel computation. Most of the applied work in this area and much of the basic research relies heavily on simulation. Problem representations, network types and topologies, training algorithms, theoretical predictions, neural network modularization, combination, and application approaches are usually explored empirically: Prototypes are built in the form of simulation programs and are then evaluated in dozens or hundreds of program runs. Since training a neural network is a computationally intensive task and neural networks contain much inherent parallelism, parallel implementations are an obvious path. In practice, however, there is lack of such implementations; only simple neural network models have been implemented on parallel machines (see Section 2).

1.1 Goals of this work

To understand this situation, we have to recognize that there are two types of users of parallel machines: Users of type A will do almost anything to get maximum performance, because their computation requirements are so extraordinarily high. Such users will be satisfied with explicit message-passing programming, which is very cumbersome but also very efficient. The other, much larger class of users, type B, is willing to trade some efficiency for ease of implementation. For regular problems, such users will often find implementations in more abstract languages such as HPF "fast enough". However, when it comes to irregular problems, no sufficient programming support for type B users is currently available; the available compilers produce extremely inefficient implementations in this case. The present work aims to provide such support.

In this work, I present a compiler that translates high-level programs for constructive neural network training algorithms. These programs change the interconnection topology of the neural network during program execution, leading to dynamic and irregular data and problem structures. The compiler generates implementations that have near-optimal data locality and load balancing with a minimum of dynamic data redistribution. This enables type B users to use parallel machines even for irregular problems. The target architectures are parallel distributed memory machines with hundreds or thousands of processors that are well-balanced with respect to communication versus computation performance, so that a fine-grained implementation can be efficient. See Section 5.5 for a discussion of other possible platforms.

1.2 The domain-specific language approach

The basic idea of my work is the following:

> *For neural network algorithms, a compiler can find inexpensive data and load distribution schemes that work well even for problems with dynamically changing structure. However, this is true only if the compiler has enough information about the semantics of the program. Semantically rich program descriptions supply such information in the form of constraints on the program behavior to be expected.*

Dynamically changing neural networks occur in the context of constructive (additive) learning algorithms and selective (pruning) learning algorithms. Such learning methods are useful for partially automating the tedious search for an appropriate network topology (primarily additive methods, but also pruning as a replacement for or complement to regularization), for fine-tuning successful topologies (pruning methods), and for reducing the resource consumption of successful topologies (pruning methods). These methods may become still more useful as the size of neural networks that can reasonably be trained increases, because training effort is superlinear in network size.

There are two approaches to such semantically rich descriptions of dynamic neural algorithms: An existing object oriented language could be extended by providing a set of predefined classes with fixed semantics and constraints on their use. Or a domain-specific special purpose language could be built. I have used the latter approach; the neural network programming language designed for this work is called CuPit, after Warren McCulloch and Walter Pitts who first described a formal neuron in 1943 [18]. For a description of CuPit see [23]. Using a special-purpose language approach avoids the enormous complexities that would arise in the implementation of compiler optimizations when extending a general object oriented language. Such complexities would arise due to interactions of the extension features with standard features such as general object references and inheritance.

Compilers for CuPit can exploit typical properties of neural algorithms: Most computations are local to objects in the network, non-local operations occur in patterns that are regular with respect to a given network topology, load is almost proportional to the amount of connections in the network, and the network topology changes only from time to time. These properties will be further elaborated in Section 3; their exploitation is explained in Section 4 (approach). The performance improvements obtained with the techniques are qualitatively presented in Sections 4 (approach) and 5 (implementation) and will be quantitatively investigated in Section 6 (results).

1.3 Generalizability

Is this work of general importance? It can be considered a case study for the use of domain-specific information in languages and compilers. This approach may provide user-friendly ways to efficient parallel implementations also in many other restricted application domains besides neural algorithms. Wherever non-computer-scientists are to produce custom parallel software for their personal computing problems, efficient domain-specific languages could provide much leverage for widespread success of parallel computing. More concretely, some of the techniques or, more importantly, ideas presented here may be usable in or transferable to other domains as well.

2 Related work

In general for parallel implementations the problems of data locality and load balancing need to be solved. Graph partitioning is one approach to solving both at once. Aspects of these three topics will shortly be discussed. More specifically, work on parallel implementations of neural networks is obviously relevant so we will also review some of that.

2.1 Data locality

Data locality can be optimized statically for languages with array-based data parallelism, e.g. [3, 21]. Index analysis is used to compute a good data distribution for programs that use mostly affine index expressions. However, irregular computations can most often not be expressed by such programs. Fortunately, in the neural algorithm domain explicit analysis of data dependencies can be avoided, because data interactions are restricted by the current connection topology of the neural network. This property can be exploited to achieve high data locality even without using sophisticated program analysis as I will show in Section 4.

2.2 Load balancing

For load balancing (in the context of data parallel programming sometimes called *loop scheduling*), there are two radically different approaches. Dynamic load balancing is the general approach: Work is distributed as necessary *during* the execution of a parallel section. A variety of methods have been proposed, see [5] for an overview. For highly irregular problems with unpredictable run time of the parts, only dynamic methods can guarantee satisfactory balance. The disadvantage of dynamic load balancing methods is that they are inherently unable to guarantee data locality, because it is impossible to predict on which processor a certain operation will be executed. Static load balancing, on the other hand, fixes the distribution of work for a parallel section before that section begins. The simplest version of this approach is implicitly taken with the data locality optimizations mentioned above: It is the assumption that the work will be balanced when data is distributed evenly, i.e., that the work to be done is the same for each data element; a similar assumption is used here for neural algorithms. In this case, static load balancing is a compile-time method, whereas dynamic load balancing is always a run-time method.

2.3 Graph partitioning

A class of methods trying to solve the data locality and load balancing problem at once is based on graph partitioning; [10] gives a good overview and references. These methods assume that the program's communication and computation graph is known in advance. Graph partitioning tries to cut this graph into a given number of parts minimizing the weighted sum of cut edges (maximizing data locality) and having roughly the same sum of vertex weights in each part (balancing the load). The parts are then distributed over the processors of the machine. The problem with these methods is that since the exact solution requires exponential time, they are all heuristic and are either quite expensive or produce poor results. The long running time of the better methods usually forbids to use them frequently at run time. Graph partitioning is also not readily applicable to neural algorithms, because graph partitioning methods assume that the program performs operations on all considered objects (here: all nodes and connections of the network) at the same time.

2.4 Implementations of neural networks

Lots of work has been performed on supporting the implementation of artificial neural networks. Quite a bit was also performed for parallel machines. However, I am not aware of research towards optimizing implementations of neural networks with dynamically changing topologies using fine-grained parallelism. Current work is mostly concerned with either highly optimized implementations of *individual* neural algorithms, usually assuming regular neural network topologies and specific machines (e.g. [7, 31, 16] and references in the latter), mapping of more general but static neural networks to specific low-latency [15] or higher-latency parallel machines (e.g. [6, 30] and references therein), implementations for special-purpose hardware (e.g. [9, 20, 27]), or more coarse-grained approaches on workstation clusters (e.g. [6, 12]). Furthermore, there is a substantial number of suggestions for *neural network description languages*. Most of them, however, describe only the static topology of a network and cannot express any actual algorithm. Of the rest, very few have been implemented on any parallel platform. The probably most advanced proposal is CONNECT [13, 14], but even this does not support dynamic changes to the network topology. See [19] for a good survey of parallel neural network platforms.

Summing up, there are currently no approaches that allow for high-level, yet efficient parallel implementation of dynamic irregular neural networks.

3 What is a neural network?

Let us define *neural networks* and *neural algorithms* as suited to our needs. Many of the definitions given below refer to certain lines in Figure 1 where fragments of a CuPit program are shown to exemplify the descriptions. Roughly the same program will also be assumed in the description of the data distribution and code generation techniques later on. Familiarity with basic neural network terms is assumed.

3.1 Neural network

A neural network is a collection of *nodes* (often called *units* or *neurons*) and directed *connections* (often called *weights*). These nodes and connections form a directed graph. The structure of this graph is called the *topology* of the network. We define neural networks in terms of data types. There are connection types, node types, node group types, and network types.

A *connection* may carry an arbitrary data structure of fixed size, determining its *connection type* (lines 1-14 of Figure 1). The data structure consists of fields called *data elements* (lines 2-5), just like a record type. A connection links two not necessarily different nodes; at one node it is an *outgoing* connection, at the other it is an *incoming* connection. At a node, a connection is attached to an *interface* (lines 19-20) that can accept only either incoming or outgoing connections of a single connection type.

A *node* may have arbitrary data elements (lines 21-22). In addition, there are a number of interfaces to the node (lines 19-20), as mentioned above, each defined by an *interface mode*

```
 1 TYPE Weight IS CONNECTION
 2   Real     i       := 0.0,
 3            o       := 0.0,
 4            weight  := 0.0,
 5            delta   := 0.0;
 6   PROCEDURE prune(Real CONST pruneThreshold) IS
 7     IF ME.i <= pruneThreshold
 8     THEN REPLICATE ME INTO 0;  END;
 9   END PROCEDURE;
10   PROCEDURE transport(Real CONST val) IS
11     ME.i := val;
12     ME.o := val*ME.weight;
13   END PROCEDURE;
14 END TYPE;

15 Real REDUCTION rsum IS
16   RETURN (ME + YOU);
17 END REDUCTION;

18 TYPE SigmoidNode IS NODE
19   IN  Weight in;
20   OUT Weight out;
21   Real       inData;
22   Real       outData;
23   PROCEDURE forward(Bool CONST doIn,doOut) IS
24     IF doIn  THEN
25       REDUCTION ME.in[].o:rsum INTO ME.inData;
26     END;
27     IF doOut THEN
28       ME.outData := activation(ME.inData);
29       ME.out[].transport(ME.outData);
30     END;
31   END PROCEDURE;
32 END TYPE;

33 TYPE Layer IS GROUP OF SigmoidNode END;

34 TYPE Mlp IS NETWORK
35   Layer   f, h, t;
36   PROCEDURE createNet(Int CONST inputs,
37                               hidden,outputs) IS
38     EXTEND ME.f BY inputs;
39     EXTEND ME.h BY hidden;
40     EXTEND ME.t BY outputs;
41     CONNECT ME.f[0].out TO ME.t[].in;
42     CONNECT ME.f[].out TO ME.h[].in;
43     CONNECT ME.h[].out TO ME.t[].in;
44     (* ...initialize weights etc. *)
45   END;

46   PROCEDURE example() IS
47     ME.f[].forward (false, true)
48     ME.h[].forward (true,  true);
49     ME.t[].forward (true,  false);
50     ME.t[].backward(false, true);
51     ME.h[].backward(true,  true);
52   END PROCEDURE;
53 END TYPE;

54 Real IO x1, x2;
55 Mlp VAR net;    (* the NETWORK*)

56 PROCEDURE program() IS
57   net[].createNet(inputs,hidden,outputs);
58   REPLICATE net INTO 1...300;
59   REPEAT
60     getExamples(x1,x2,REPLICATES(net));
61     net.f[].inData  <-- x1;
62     net.t[].outData <-- x2;
63     net[].example();
64     (*...merging,weight update,etc.*)
65   UNTIL stopTraining()  END REPEAT;
66 END PROCEDURE;
```

Figure 1: Fragments of an example CuPit program. The procedures `activation, backward, getExample`, and `stopTraining` are not shown.

(either "incoming" or "outgoing") and a connection type. Data elements and interfaces together determine a *node type* (lines 18-32).

Nodes are aggregated into *node groups*. Objects of a particular node group type (line 33) can consist of zero or more nodes of that node type.

A *network* may have arbitrary data elements (not shown in the example). In addition, a network type declares a fixed number of node groups (line 35) to be part of the network type (line 34-45).

Note that this definition rules out symmetric network types (like Hopfield networks) that require the connections to be undirected. In other respects, though, the model is quite flexible, as one program can contain an arbitrary number of types.

3.2 Neural algorithm

A *neural algorithm* is a program that manipulates such a neural network with operations of only the following kinds: A sequential program called the *central agent* (lines 56-66 of Figure 1) that controls the learning algorithm can

- read and write data from and to the nodes of the network (lines 61-62) using I/O buffers (lines 54, 60) and
- call *network procedures* (lines 57, 63).

Network procedures can

- manipulate the data elements of the network object (not shown),

- call *node procedures* to be executed for all or some of the nodes of a particular node group that is part of the network (lines 47-51),
- create or delete nodes in a node group (lines 38-40),
- create or delete connections between a particular pair of interfaces of some or all of the nodes of two node groups (lines 41-43), and
- compute a reduction over a particular data element of some or all nodes of a node group using an arbitrary reduction operator (not shown).

Node procedures can

- manipulate the data elements of the node object (line 28),
- call *connection procedures* to be executed for all of the connections attached to a particular interface of the node (line 29),
- delete the node they are applied to or create multiple copies of the node (including cloning of all the connections, not shown), and
- compute a reduction over a particular data element of all connections attached to a particular interface using an arbitrary reduction operator (line 25).

Connection procedures can

- manipulate the data elements of the connection object (lines 11-12), and
- delete the connection object they are applied to (line 8, the `prune` procedure could be called, indirectly, from line 64).

Note that calls to node procedures and connection procedures imply parallelism ("group calls"). Network procedures, node procedures, and connection procedures operate only on the local data elements of the particular network, node, or connection object they are applied to and on the parameters that are supplied with the call. Such parameters are read-only.

In addition, we define *network replication* to mean the following:

- Creating network replicates (line 58) means to make identical copies of a network;
- merging network replicates (not shown) means to unify the data in all networks in a set of replicates by means of user-defined, elementwise reduction operations and redistribution of the results;
- deleting network replicates (not shown) means to create a single network from a set of replicates by merging without redistribution;
- executing a network operation on replicates (line 63) means to execute the operation for each replicate using the same procedure but different training examples.

Replicates can identify themselves by a replicate number. While several replicates of a network exist, the topology of these networks may not change, because this could lead to diverging topologies, which can not uniquely be merged again. With network replication, calls to network procedures imply parallelism, too (line 63).

Network replication essentially implies parallelism on the level of training examples in a neural algorithm. This kind of parallelism is applicable to many but not all neural algorithms.

The above formulation of neural algorithms gives three nested *levels of parallelism*: The sequential program invokes a network procedure on several network replicates in parallel, a network procedure invokes a node procedure on several nodes in parallel, and a node procedure invokes a connection procedure on several connections in parallel.

The explicit modeling of the network, node, and connection types and their operations in CuPit enables a compiler to access most information required for optimizations quite easily. In particular, the compiler straightforwardly knows which kinds of operations will be applied to which objects and thus can use appropriate data and process distributions, as described in the following section.

4 Approach

The kernel of a typical neural algorithm consists of repeatedly putting a training example into the network and then propagating it through the nodes and connections of the network one or several times. Most time is spent in the broadcast from nodes to connections, reduction from connections to nodes, and local operations in nodes and connections. Note that what is called reductions above will actually be a number of reductions at once, one for each node.

In this context, the following considerations lead to the maximization of data locality: (1) Local operations can be forced to have full data locality by attaching the computation to its data object. (2) Reductions cannot have full data locality in any useful parallel implementation; they can, however, exploit both block-wise locality and neighborhood relations in the communication network of the parallel machine. (3) Many broadcasts can be avoided using data replication and additional computation.

The following consideration leads to load balancing on the relevant (i.e., connection) level: For each call of a connection operation, each processor should hold approximately the same number of connections on which the operation works. This simple rule is sufficient, because for each parallel connection operation the work to be performed is nearly the same for each connection, since connection operations usually contain no loops. Note that the data locality and load balancing goals conflict.

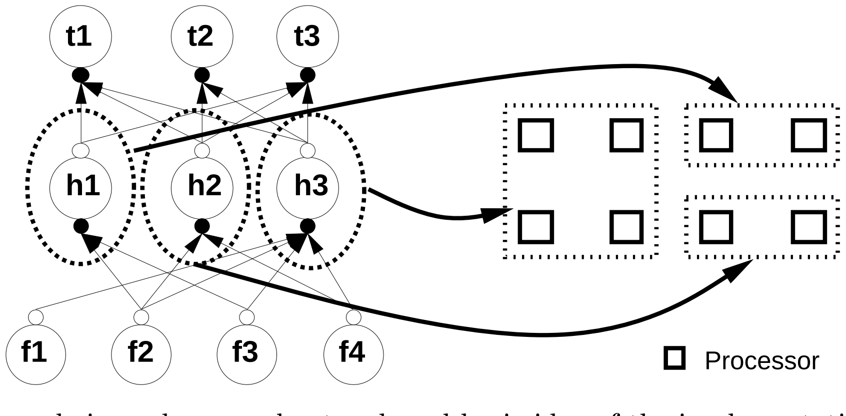

Figure 2: Example irregular neural network and basic idea of the implementation approach.

The above considerations lead to the implementation approach schematically shown in Figure 2. The basic idea is to reserve a particular block of processors (or a fractional processor in more coarse-grained cases) for a node *plus* all its connections and choose the size of that block proportional to the amount of work to be performed on the connections.

The figure will become fully understandable only later on. Please refer back to it whenever necessary for keeping the global picture. To realize the approach of Figure 2, the problems shown in Figure 3 have to be solved. Both the approach and the individual problem solutions will be described in detail below.

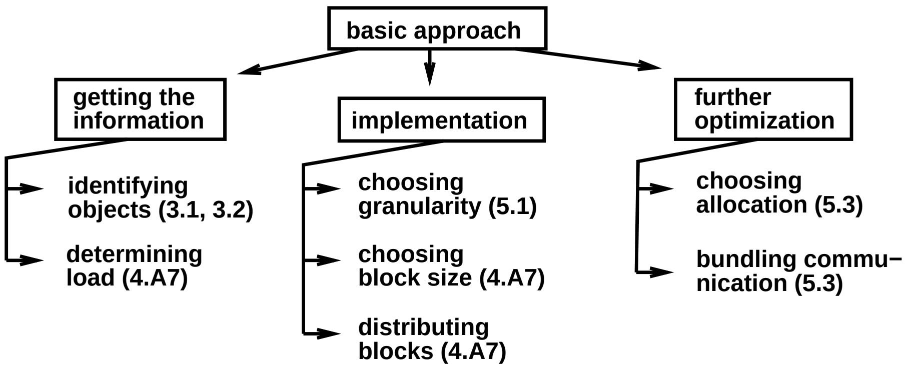
basic approach
getting the information
implementation
further optimization
identifying objects (3.1, 3.2)
determining load (4.A7)
choosing granularity (5.1)
choosing block size (4.A7)
distributing blocks (4.A7)
choosing allocation (5.3)
bundling commu–nication (5.3)

Figure 3: Subproblems to be solved to implement the approach. The numbers in parentheses indicate in which section the problem is discussed.

The approach is suitable for implementation on any distributed memory parallel machine, SIMD or MIMD. For simplicity of description, in the rest of the text we will assume a 2-D grid as the interconnection network and a rather large number of processors. Other topologies can be handled analogously, small numbers of processors require to allocate fractional processors. See also Section 5.5.

The approach taken in this work to combine all of the above considerations is the following:

A1, training example parallelism: Partition the machine into 2-rectangular *segments* of several processors each and use one such segment for each network replicate, if any; a segment is *2-rectangular* iff it is rectangular with the height being a power of two and the width being either the same as the height or twice the height, all measured in number of processors. This form minimizes the diameter while making address computations simple and fast. See also A6. All segments have same size and contain exactly the same data structure, only with different values. Using replicates trades additional work for replicate creation (once) and replicate merging (repeatedly) for increased parallelism and a reduction of the average communication distance.

A2, node parallelism: For each node group, partition the segment into 2-rectangular *node blocks* of one or several processors. Allocate one node block for each node. See Figures 2 and 4 for an example of segment and node block partitioning. How the partitioning is actually computed will be described in A7 below.

A3, connection parallelism: For each interface of each node, distribute evenly the connections of the interface over the node's block. See Figure 5 for an example of connection distribution over node blocks.

A4, data locality: Use the owner-computes rule, so that local operations on networks, nodes, and local connections always have full data locality. To get data locality for the parameter broadcast of calls that introduce additional parallelism, replicate scalars on all processors of the machine, replicate the data elements of networks on all processors of the network's segment, and replicate the data elements of nodes on all processors of the node's block. This data replication costs one machine-wide broadcast per network

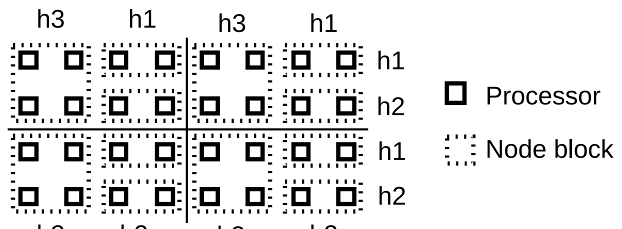

Figure 4: Example segment and node block layout of node group h of the example network from Figure 2 on a 4x8 processor grid using 4 replicates. Segment boundaries are indicated by solid lines.

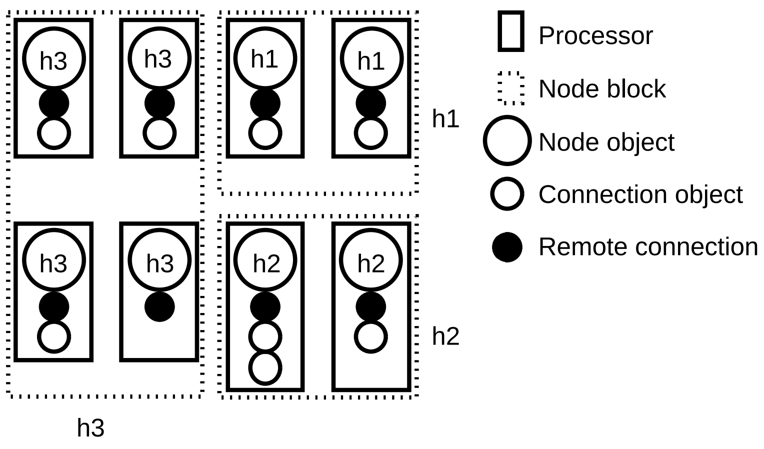

Figure 5: Distribution of node, connection, and remote connection objects of node group h of the example network from Figure 2 within one segment from Figure 4.

procedure call, a broadcast of the result after each reduction, and some memory. The corresponding broadcast savings are at least equivalent to the above broadcast costs. More savings result if the user program computes additional parameters locally in a node. See A8 for a discussion of locality of multiple nodes.

A5, minimize communication frequency: Do not replicate connection data at both ends of a connection. Writes are almost as frequent on connections as are reads, hence the savings on reads that arise due to replication would be eaten up by re-replication cost after writes. Instead, one end of a connection (called the *remote end*) contains only a pointer to the other end (called the *data end*); the data end contains a pointer to the remote end plus the actual connection data object. For best efficiency, the correct decision has to be made at which end to put the actual data. Communication for operations on remote connections can easily be aggregated. See Section 5.3 for details.

A6, minimize communication distance: To make reductions of connection data into nodes cheap, choose node blocks to be small-diameter sets of processors. Small diameter also speeds up data replication in a node block, which is necessary after a reduction. For instance in Figure 4, the block $h3$ is sized 2×2 instead of 4×1. 2-rectangular blocks are good in this respect: they have minimum diameter for up to 256 processors (and are for instance only 5.5% larger than the minimum for 8192 processors). For other topologies than grids, analogous definitions of *2-rectangular* can be found.

A7, load balancing: Make the size of each node's block proportional to the work performed on the connections attached to the node. Work can easily be directly measured at run time. When different interfaces of a node have a different number of connections, node block size can only on the average be proportional to the amount of work. The algorithm given below can be used to compute the node block sizes $b_1 \dots b_k$ of k nodes having connection work equivalents of $w_1 \dots w_k$, respectively, for a segment of S processors. In addition to the algorithm given, node blocks must be forced to have at least size 1. The problem solved by the algorithm is to start with the optimal node block size proportions given by the work proportions and to compute 2-rectangular node block sizes that minimize slack yet maintain maximal faith to these optimal node block size proportions. Let $p(n)$ mean 2^n.

Node block size computation algorithm:

```
    $W := \sum_{i=1}^{k} w_i$;              (*total work*)
    Forall $i \in [1 \dots k]$ :
        $s_i := S \cdot w_i / W$;           (*theoretical fractional node block sizes*)
        $u_i := p(\lceil \log_2 s_i \rceil)$;     (*2-rectangular sizes obtained by rounding up*)
        $d_i := p(\lfloor \log_2 s_i \rfloor)$;     (*2-rectangular sizes obtained by rounding down*)
        $r_i := u_i / s_i$;                (*rounding ratios*)
    by binary search approximate the maximum $\alpha \in [1 \dots 2]$ so that
        $\sum_{i, r_i < \alpha} u_i + \sum_{j, r_j \geq \alpha} d_j \leq S$
    (*now we have the best block sizes using a fixed splitting point $\alpha$*)
    (*we minimize slack by rounding some more blocks up instead of down:*)
    now let $J$ be the sequence of indices $j$ from above and
    $\bar{J}$ a leading subsequence of $J$. Find the maximal $\bar{J}$ that maintains
        $\sum_{i, r_i < \alpha} u_i + \sum_{c \in \bar{J}} u_c + \sum_{j \in J \setminus \bar{J}} d_j \leq S$
    and, using the above index sets of $i, c$ and $j$, set
        $b_i := u_i$,   $b_c := u_c$,   $b_j := d_j$ for all $i, c, j$ from above
```

From this set of node block sizes the actual node block layout is computed by a bin packing algorithm. The constrained version of the bin packing problem assuming 2-rectangular bins and pieces can be optimally solved in time proportional to the size of the bins and is parallelizable to logarithmic time. An example result of these two algorithms is shown in Figure 2.

A8, node co-locality: Attempt no optimization of remote connection locality since it does not pay off. We could arrange the node blocks within a segment and the connections within a node block in a way that maximizes remote connection locality, i.e., that results in having both ends of a connection on the same processor as often as possible. There are two reasons for not doing this: First, little such locality can be obtained in neural networks, since their topology typically exhibits almost no clustering of connections. (An exception are modular neural networks, for which a "locality preference" for the connections holds, e.g. [30]). With only small gains in locality, the long run time of the optimization computation takes too long to amortize; see the end of Section 6.1 for a quantification of this argument. Second, arrangement of nodes for optimal extra-object locality interferes with arrangement of node blocks for minimum waste of processors within a segment. Hence, we either have to trade extra-object locality for processor utilization or have to give up using 2-rectangular node blocks. The latter would make the layout algorithm expensive. See the end of Section 6.2 for a quantitative justification of this crucial decision.

5 Implementation

The implementation discussed here is for a MasPar MP-1 (Model 1216A). The MP-1 is a 16384 processor SIMD machine that is a good basis for this work because of two properties: First, due to the large number of processors scaling behavior is explored properly. Second, the machine's communication performance is well balanced with its computation performance. For the mix of communication operations found in neural algorithms each floating point communication takes about as long as 5 to 20 floating point multiplications (including loads and stores). Most machines today are very much slower, but future machines may be better by using latency hiding techniques and appropriate faster network hardware and software.

The CuPit compiler was implemented using the Eli compiler construction system [8] and generates MPL [17] code, MasPar's data parallel variant of C. The compiler is used much like a normal C compiler. With the exception of choosing the number of replicates (which is not currently automated), all optimizations are applied fully automatically; no user intervention in the form of annotations or selection of compiler options is required. The source code of the compiler is available as a literate programming document [25]. Some details of the implementation will be described in the following subsections.

5.1 Networks

Network replicates are created only on request from a user program. After topology changes, the network data structure is completely reorganized when replicates are created again or upon program request. Calls to network procedures are executed on all processors of the segment.

CuPit lets the programmer specify the number of network replicates as an interval; the program may choose any number of replicates from this interval at run time. The best value with respect to execution time depends on the current size of the network, the number of training examples in the dataset, the size of the machine, and the training algorithm used. Therefore, the only practical way to find good choices for this parameter automatically is to generate code that makes the program search iteratively, at run time, for optimal values, using changes in run time per training iteration to trigger the search.

5.2 Nodes

Calls to node procedures are executed on all processors of the node blocks of the participating nodes. No broadcast of parameters is necessary, since these are locally available due to network data element replication. An exception are input or output of training examples into or from nodes, performed by special CuPit operators that are called from the sequential part (central agent) of the program and that act on all nodes of a node group in all network replicates at once. The input operator implies broadcast over the node block. Another special case are reduction operations over the connections of a node. Such reductions return their result in the first processor of each node block; it is then immediately broadcast to all processors of the node block in order to maintain data replication.

5.3 Connections

Within each node block, the connections attached to the node are distributed evenly on a per-interface basis. For each connection type, a decision is made as to whether the data end of the connections is located at the input interface or the output interface. When using the recommended CuPit coding style it is obvious for any given program which is the right decision. Therefore, the decision is fixed in the compiler but can be changed via a compiler switch if necessary. Calls to connection procedures are executed on all processors of the node blocks of the nodes issuing the call, so no broadcast of parameters is necessary.

The CuPit compiler computes the sets of elements to be fetched and sent at the beginning and end of remote calls for each connection procedure by a very simple conservative static analysis. The criterion used is textual presence of an element in any read (right hand side) or write (left hand side) position, respectively, somewhere in the static call chain of the procedure. This criterion works well for typical neural algorithms and need not be replaced by sophisticated data flow analysis. The elements to be fetched or sent are not communicated individually but are aggregated into packets that minimize communication time. On the MasPar, this means to aggregate all elements that have gaps of less than 12 bytes between them into one packet and to transfer also the gaps instead of starting a new communication; packing and unpacking is not feasible in reasonable time on this machine but may be appropriate on others.

Figure 5 shows the data distribution inside the node blocks of Figure 2 for one segment. Each processor contains one node object copy, plus zero or more connection objects, plus zero or more remote connection objects. Connection objects correspond to outgoing connections, while remote connection objects correspond to incoming connections. The block size has been chosen in proportion to the amount of connection work to be done by each node, as shown in A7. This amount is a weighted sum of the numbers of incoming and outgoing connections; the weights used are the average amount of work to be done for each connection kind. This amount is usually higher for remote connections, since sending and fetching remote connection parts takes a significant part of overall connection operation run time.

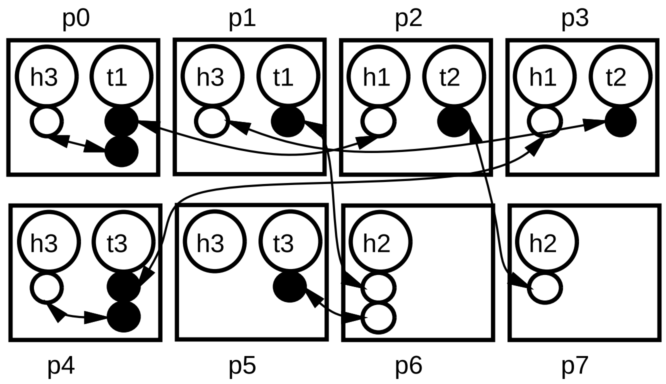

Figure 6: Layout of nodes of node groups *h* and *t* of the example network and the connections between them. Each connection is implemented as a pair of a connection object and a remote connection object, each having a pointer to its counterpart.

Figure 6 shows the distribution of connections and corresponding remote connections between node groups h and t from Figure 2 for one segment. The distribution of t shown in the figure uses the straightforward node block sizes; in contrast to these, the node block size computation algorithm given above would assign 4 processors to $t1$ or to $t3$, thereby spreading their connections more evenly and thus improving overall communication performance.

5.4 Miscellaneous details

All objects carry along descriptors that indicate their validity status (existence) and other data as needed: Networks know their replicate number and segment size. Node groups know their number of nodes and local array size. Nodes know their index and their block size. Connection interfaces know their local array size, number of connections, and amount of work performed per connection. Connections know the location of their opposite end.

Self-deletion of connections and nodes is done by setting the existence indicator in the respective descriptors to false; the actual removal of the data objects occurs at the next reorganization of the data structure during a `REPLICATE` call. Creation of nodes is done by instant reorganization of the node group. There is always an exact one-to-one correspondence of the data structures in each segment, thus replicate merging can be computed easily.

5.5 Implementing CuPit on other platforms

One might ask whether or how the techniques described above apply to parallel implementation platforms other than SIMD machines. The most important instances of these are MIMD machines with either message-passing or bus-based shared memory (SMPs)

For MIMD message-passing platforms the central difference is one of granularity. The number of processors is comparatively much smaller and the ratio of computation to communication performance much larger. Hence, a fine-grained distribution of the neural network based on the individual connections would be inefficient. It will not be used and all related techniques are no longer needed. Given that, however, some simple form of adaptive node group partitioning needs to be used in order to achieve load balancing for irregular networks. The other ideas described above still apply in similar form. In particular, communication aggregation can be extended to accommodate messages from multiple nodes in a single packet.

The issues when implementing CuPit on symmetric shared-memory multiprocessors (SMPs) were investigated in our work on CuPit-2. The results are described in [11]. Roughly speaking, similar issues of granularity come up as for MIMD message-passing. In addition, optimizing cache performance becomes a central task in order to avoid contention of the central bus.

6 Results and discussion

The effectiveness of the compiler optimizations was evaluated in various ways.

1. To measure the improvements achieved by load balancing, a series of experiments was run with the same programs with and without load balancing.
2. To estimate the savings due to data locality, code with additional communication operations to simulate non-local data distribution was generated and its run time was compared with the optimized code. The cost-performance ratio of graph partitioning methods for still better data locality was also assessed empirically.
3. To estimate the cost of computing and creating the data distribution that leads to data locality and load balancing, the fraction of time spent in data distribution procedures was measured.
4. To estimate how good the overall performance is, a comparison with the code generated by an optimizing compiler for a general-purpose high-level language was made.
5. One experiment assessed the relative speed obtained by making the optimal versus the non-optimal decision for connection allocation.
6. To estimate the usefulness of communication aggregation, programs with aggregation were timed against programs that fetched each element individually.
7. One experiment assessed the relative speed obtained by dynamic adaptation versus static choice of the number of network replicates.

These experiments and their results are described and discussed in the following sections.

How to use CuPit for actual programming is discussed in some more detail in [26]. That reference allows for better judgment of the adequacy and ease of use of the language.

6.1 Load balancing

For the load balancing experiments, irregular network topologies were created by a network pruning algorithm. Such algorithms start with a large, fully connected network and remove some of the connections in several pruning steps during the training process. Which connections to remove was decided using a statistical measure of significance that the weight is non-zero (*autoprune* method, see [4] for a detailed description). As experiments showed, the actual pruning criterion is far less important for the results than the dataset used to train: Some datasets show significantly higher irregularity in pruned networks than others. The higher the network irregularity, the more performance is gained by load balancing.

Training started with 4-layer networks with 20+20 hidden nodes and all possible feed forward connections, including all shortcut connections. To ensure comparability, a static pruning schedule was used: prune 30%, 15%, 15%, 15%, and 15% of the remaining weights after epoch 40, 80, 120, 160, and 200, respectively. Altogether, this schedule prunes about two thirds of the initial connections. Although such a static pruning schedule is not the way pruning would be used in a real application, it is sufficiently close to real pruning schedules for our timing measurement purposes. Note that the measurements are conservative: In real network pruning, often more than 80 or 90 percent of the connections are removed; load balancing would be even more effective then, compared to the results shown below.

For the experiments reported here, 11 real training problems from 10 different domains were used, all taken from the Proben1 benchmark set [24]. The name and size of

Problem	N_{in}	N_{out}	N_{ex}	dbal	nbal	noloc	cwrong	comm	repl
building	14	3	2104	102	123	244	145	110	99
flare	24	3	533	105	120	289	141	112	165
hearta	35	1	460	102	114	325	151	111	110
cancer	9	2	350	110	150	305	144	97	98
card	51	2	345	102	139	333	164	108	120
diabetes	8	2	384	108	129	294	159	110	161
gene	120	3	1588	102	115	221	149	119	77
glass	9	6	107	114	130	309	165	101	102
heart	35	2	460	105	130	320	153	111	110
soybean	82	19	342	105	132	288	167	115	130
thyroid	21	3	3600	100	120	247	144	121	114
(average)	34	4.2	934	105	127	289	154	110	115

Figure 7: Problem sizes and relative run time of non-optimized program versions. N_{in}, N_{out}, N_{ex}: Number of input nodes, output nodes, and training examples, respectively. dbal, nbal, noloc, conall, comm, repl: Relative run time (in percent) of dumb load balancing, no load balancing, no data locality, wrong connection object placement, no remote connection access communication bundling, and static choice of number of network replicates, respectively, compared to optimized version for various data sets in a network pruning situation.

each problem is given in the first four columns of Figure 7. For all these problems, three different versions of the pruning program were timed: An optimized one with load balancing based on actual measurements of workload at each connection (called *bal*), one with load balancing based on mere connection counting as opposed to load measurement (called *dbal* for "dumb balancing"), and one without load balancing (called *nbal*, for "no balancing"). The cost of the actual run-time load measurement is negligible on the MasPar. The timings reported here are based on the time needed for training in epoch 210, i.e., after the last pruning step. The values are normalized so that *bal* is always 100.

The results appear in Figure 7 (ignore the rightmost columns for now, they will be described in later sections). Summing up, we find an average relative run time for load balancing based on load estimation instead of load measurement of 105% and for unbalanced load of 127%. Note that the latter value is a very conservative estimation of the effect of load balancing, for two reasons. First, on the MasPar, communication latency is extremely low for low communication traffic. Hence, communication gets faster almost in proportion to the reduction of traffic as it happens in programs with misbalanced load and reduces the effect of load misbalance. Other machines are less friendly in this respect. Second, the irregularities in the networks of the example runs were only moderate, since only two thirds of the connections had been pruned. Therefore, the potential for speedups from load balancing was only moderate, too. Thus, for most situations we can expect the actual effects of load balancing to be higher than the 27% mentioned above.

Additional experiments were performed in order to estimate the effects of load balancing for machines that perform latency hiding. The compiler was instrumented to generate code that simulates such machines by completely ignoring time used for communication of remote connection data in timing measurements as well as in load balancing computations. We might expect load balancing to be less effective in this situation. Experiments with the *gene* data sets showed, however, that the decrease in effect of load balancing was minor:

The relative run time of the latency hiding program without load balancing compared to that with load balancing was 113%. This effect of load balancing is less than 2% weaker than for the normal MasPar implementation. Hence, even for machines that have quasi-zero latency, performance gains due to load balancing will be significant.

Under the assumption that network pruning leads to topologies that have a typical degree of irregularity, I conclude that load balancing uniformly can save at least about one fifth of overall run time on a variety of machines.

6.2 Data locality

It is not quite clear with which alternative implementation to compare code generated by the compiler for an evaluation of the effect of data locality. I chose an array-based implementation with regular array distribution. The connections of irregular networks could be stored in such arrays as follows: The set of all connections attached to one interface of all nodes of one node group are densely stored in one array. Each node has a pair of indices indicating the part of the array where its connections are stored. Such a scheme would have remote connection access for all connection operations. Furthermore, node data replication can no longer be used to avoid broadcast of the parameters of connection operations. On the other hand, this array-based implementation has a better memory utilization and always has perfectly balanced load. This approach to irregular problems is used by languages such as NESL [2].

In order to estimate the impact of such array-based implementations on performance, the CuPit compiler was instrumented to generate code that simulated having no connection object data locality for connection operations (but still avoided parameter broadcast, thus again leading to a conservative estimate). Timing measurements with these non-data-local variants of the otherwise unchanged program produced the results indicated in the *noloc* column of Figure 7. Note that the time for merging network replicates is excluded in these values, which is equivalent to measuring with very large training sets. This correction was made because the replicate merging code generated by the modified compiler did not ignore data locality, which would have influenced the results. As we see, implementations without data locality take about two to three times as long to execute.

Further experiments (performed using the the Chaco [10] program) showed that using graph partitioning methods (Kernighan-Lin heuristic and spectral bisection or octasection) to compute the data distributions would not pay off: After pruning two thirds of all connections graph partitioning could increase remote connection locality only by about 10%. The following numbers depend heavily on the actual network size and structure: with graph partitioning we can expect savings in run time on the order of typically between 0% and 15% due to increased connection locality, depending on the amount of irregularity. However, these savings would be outweighed by the heavily increased cost for the data distribution procedure of about additional 4% to 22% run time, because the current cost of 2% to 11% of overall run time for the data distribution would roughly triple.

6.3 Cost of data distribution

The above results all exclude the time spent in the general data distribution procedure that are able to achieve data locality and load balancing. But doing without data locality or without load balancing might allow for simpler and faster data distribution procedures. Therefore, using the same experimental setting as in Section 6.1, I measured how much time was spent in the general data distribution procedures: Depending on the size and structure of the network and the number of replicates, the data distribution procedures accounted for 2 percent (gene problem) to 11 percent (glass problem) of run time. This includes the initial creation of replicates after the construction of the network and the deletion and recreation of replicates before and after each pruning step.

As we see, the cost of data distribution (of which only parts could be saved) is significantly smaller than the gains from load balancing let alone data locality. This is true even for problems with extremely small data sets such as the *glass* problem where there is little training time to amortize data distribution costs. I conclude that the data distribution described in this work is not only effective but also efficient in accelerating program execution.

6.4 Overall performance

To justify all other evaluations, we must be sure that the compiler produces code that is reasonably efficient, since otherwise large improvements would not mean much. For this purpose, I compared the run time of a CuPit program to an equivalent Modula-2* (pronounce: Modula-two-*star*) program. The latter was translated by a compiler that also targets the MasPar[1] and that is known to generate quite efficient code [22]. The problem chosen was backpropagation using the RPROP learning rule [28] for a fully connected 3-layer feed forward network.

The best was done to ensure that the code generated by the Modula-2* compiler was as efficient as possible: A regular network was used, since that (and only that) made the Modula-2* compiler generate code having data locality; procedure calls on the node and connection level were inlined in order to avoid the cost implied by the copy-in-copy-out semantics of array parameter passing in Modula-2*; all levels of parallelism were unrolled into a single FORALL statement to minimize startup costs of parallel sections; remote data read more than once during one operation was buffered in local variables.

Two disadvantages remained for the Modula-2* code: The code generated for the FORALL is more general than that used by the CuPit compiler to start parallel sections, and the Modula-2* compiler is not capable of combining multiple communication operations for remote connection access. On the other hand, the Modula-2* program had two advantages over the CuPit program: First, its hand-written code does of course not copy unnecessary data upon redistribution of the network data after a network replicate merge operation. This optimization is not implemented in the CuPit compiler. Second, it fetches and sends only those elements of a remote connection that are really used at

[1]The Modula-2* compiler was the only one available on the MasPar for any general-purpose parallel language that completely hides machine architecture and communication operations from the user — and hence provides a language level comparable to that of CuPit.

run time while CuPit code fetches all elements that *may* be used as determined by a very simple static analysis.

Timings were taken for runs of the following problems: a 128:13:127 network (that means 128 input nodes, 13 hidden nodes, and 127 output nodes) with 127 training examples using 64 or 16 replicates, a 129:13:128 network with 128 training examples using 16 replicates, and a 501:13:500 network with 500 training examples using 16 or 4 replicates. These problems were chosen to put the CuPit code at significant disadvantage: 13 node blocks of equal size cannot be distributed well over a 2-rectangular segment, and the small number of training examples emphasizes the overhead in redistribution after merge. Thus, the results obtained will, again, be rather conservative.

The results indicate that Modula-2* code may be faster than CuPit code when many replicates are used: for the 64 replicates example, the relative run time of the Modula-2* program compared to the CuPit program was 90%. This result is due to the savings during data redistribution after network merge. For smaller numbers of replicates, CuPit code was always faster despite the difficult conditions chosen. Over all examples, the average relative run time of the Modula-2* program was 142%. Even when the ability of the CuPit compiler to combine multiple fetch or send operations was switched off, the average was still 130%.

As we see, the CuPit code is roughly one third faster than that generated by a known-to-be-efficient general purpose parallel compiler. This result suggests that the overall quality of the code generated by the CuPit compiler is good. It must be emphasized that this test was done on static, regular problems that the Modula-2* compiler is well suited for but that are not the typical domain of the CuPit compiler which targets dynamic, irregular problems. No useful comparisons with other compilers could be performed for irregular problems, because no compilers that optimize for irregular problems are available on the MasPar.

6.5 Connection location

A decision must be made by the compiler where to locate the actual connection objects: at the input interface or at the output interface (see Section 5.3). An experiment explored the results of making the wrong decision in this respect. The experimental setup was as in Section 6.1; a program called *cwrong* that placed connection objects at output interfaces (which is the wrong decision for this program) was timed. The results are shown in the respective column of Figure 7.

As we see, a significant performance penalty of about 50 percent run time increase can be avoided by the ability to choose the better connection location. Note that this value depends on the actual algorithm of the user program and on the way it is coded.

6.6 Communication aggregation

As is shown in column *comm* of table 7, not having communication aggregation (see Section 5.3) costs an additional average 10% run time on the MasPar. The value would be higher for machines with higher latency-to-bandwidth ratio.

6.7 Selecting the number of replicates

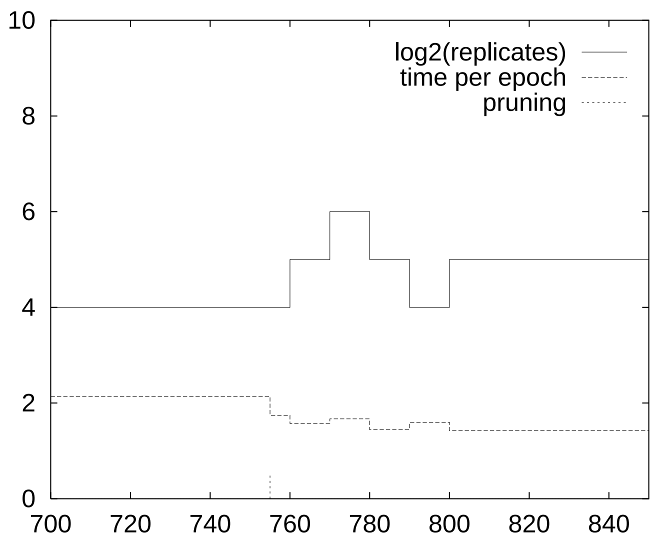

Figure 8: Automatic replicate number optimization: The horizontal axis shows the iterations through the training set ("epochs"). The curves show changes in number of replicates and corresponding run time (in seconds) per epoch initiated automatically after the pruning step in epoch 755.

Dynamically adaptive search for the optimal number of network replicates (see Section 5.1) was also timed using a pruning algorithm with various networks and data sets and compared with a static number chosen by educated guess. Figure 8 shows an example of how the adaptive search works. The relative performance using dynamic search (static search normalized to 100) is shown in table 7 in column *repls*. As we see, something can be gained in most cases. The bad result on the *gene* problem is due to the too simple-minded prototype implementation of the search method, which always uses complete epochs in each step, even for large training sets.

7 Conclusion

This work considered the problem of compiling neural algorithms formulated in a problem-oriented and machine-independent parallel language. These neural algorithms describe data parallel computations on a dynamically changing irregular neural network. The article described an approach to compiling such programs into code that exhibits near-optimal data locality and load balancing; this is an example of using domain-specific constraints for performing optimizations that would otherwise be infeasible.

Over a variety of irregular problems, a prototype implementation of the approach produced the following performance improvements: 27% due to load balancing, 195% due to data locality, and 54% due to optimal remote connection object placement. The corresponding data distribution computations consumed 2% to 11% of the time needed for the user program computations. Even for regular problems, the code generated by the prototype compiler was shown to be as fast as that of a good optimizing compiler for a general-purpose high-level parallel language.

I conclude that in the domain of neural algorithms an optimizing compiler can automatically produce efficient code for irregular problems from a high-level description. The principles of the approach should also be applied to machines with smaller numbers of processors and be compared to other techniques. Furthermore, the general idea of exploiting domain-specific constraints should also be applied to parallel computing in other application fields.

References

[1] J.A. Anderson and E. Rosenfeld, editors. *Neurocomputing: Foundations of Research.* MIT Press, Cambridge, MA, 1988.

[2] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a nested data-parallel language. *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM SIGPLAN Notices*, 28(7):102–111, July 1993.

[3] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data parallel programs. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 16–28, Charleston, SC, January 1993.

[4] William Finnoff, Ferdinand Hergert, and Hans Georg Zimmermann. Improving model selection by nonconvergent methods. *Neural Networks*, 6:771–783, 1993.

[5] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring — a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[6] Ben Gomes. *A Framework for Mapping Connectionist Networks onto Parallel Machines.* PhD thesis, EECS Dept., University of California, Berkeley, May 1997.

[7] Kamil A. Grajski. Neurocomputing using the MasPar MP-1. Technical Report 90-010, MasPar Computers, Sunnyvale, CA, 1990.

[8] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[9] Dan Hammerstrom. The CNAPS architecture. Adaptive Solutions Inc., Beaverton, OR, January 1993.

[10] Bruce Hendrickson and Robert Leland. The Chaco user's guide, version 1.0. UC-405 SAND93-2339, Sandia National Laboratories, Albuquerque, NM, October 1993.

[11] Holger Hopp and Lutz Prechelt. CuPit-2: A portable parallel programming language for artificial neural networks. In A. Sydow, editor, *Proc. 15th IMACS World Congress on Scientific Computation, Modelling, and Applied Mathematics*, volume 6, pages 493–498, Berlin, Germany, August 1997. Wissenschaft & Technik Verlag.

[12] Christian Jacob and Peter Wilke. A distributed network simulation environment for multiprocessing systems. In *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*, pages 1178–1183, Singapore, 1991.

[13] Gerd Kock and Thomas Becher. Mind: An environment for the development, integration, and acceleration of connectionist systems. In *[29]*, pages 499–504, 1997.

[14] Gerd Kock and N.B. Serbedzija. Artificial neural networks: From compact descriptions to C++. In *Proc. Intl. Conf. on Artificial Neural Networks*, 1994.

[15] Detlef Koll, Martin Riedmiller, and Heinz Braun. Massively parallel training of multi layer perceptrons with irregular topologies. In *Proc. Intl. Conf. on Artificial Neural Networks and Genetic Algorithms (ICANNGA)*, Ales, France, 1995. Springer Verlag.

[16] Xiao Liu and George L. Wilcox. Benchmarking of the CM-5 and the Cray machines with a very large backpropagation neural network. Technical Report 93/38, University of Minnesota Supercomputer Institute, Minneapolis, April 1993.

[17] MasPar Computers, Sunnyvale, Calif. *MPL Language Reference Manual*, 1990.

[18] Warren McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943. Reprinted in [1].

[19] Manavendra Misra. Parallel environments for implementing neural networks. *Neural Computing Surveys*, 1:48–60, 1997.

[20] Silvia Müller and Benedict Gomes. A performance analysis of CNS-1 on sparse connectionist networks. Technical Report TR-94-009, International Computer Science Institute, Berkeley, CA, February 1994.

[21] Michael Philippsen. Automatic alignment of array data and processes to reduce communication time on DMPPs. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 156–165, Santa Barbara, CA, July 1995.

[22] Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, 28(8):99–108, August 1993. Also as report 14/93, Fakultät für Informatik, Universität Karlsruhe.

[23] Lutz Prechelt. CuPit — a parallel language for neural algorithms: Language reference and tutorial. Technical Report 4/94, Fakultät für Informatik, Universität Karlsruhe, Germany, January 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-04.ps.gz on ftp.ira.uka.de.

[24] Lutz Prechelt. PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, Germany, September 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-21.ps.gz on ftp.ira.uka.de.

[25] Lutz Prechelt. The CuPit compiler for the MasPar — a literate programming document. Technical Report 1/95, Fakultät für Informatik, Universität Karlsruhe, Germany, January 1995. ftp.ira.uka.de.

[26] Lutz Prechelt. A parallel programming model for irregular dynamic neural networks. In W.K. Giloi, S. Jähnichen, and B.D. Shriver, editors, *Proc. Programming Models for Massively Parallel Computers*, pages –, Berlin, Germany, October 1995. GMD First, IEEE CS Press. By accident the article was *not* printed in the proceedings volume, but see http://wwwipd.ira.uka.de/~prechelt/Biblio/.

[27] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Brüls, M. Weßeling, E. Sicheneder, J. Gläß, A. Wurz, and R. Männer. Synapse-1: A high-speed general purpose parallel neurocomputer system. In *Proc. 9th Intl. Symposium on Parallel Processing (IPPS'95)*, pages 774–781, Los Alamitos, CA, April 1995. IEEE Computer Society Press.

[28] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, April 1993.

[29] Achim Sydow, editor. *15th IMACS World Congress on Scientific Computation, Modelling, and Applied Mathematics*, volume 6, Application in Modelling and Simulation, Berlin, Germany, August 1997. Wissenschaft und Technik Verlag.

[30] Tom Tollenaere and Guy A. Orban. Decomposition and mapping of locally connected layered neural networks on message-passing multiprocessors. *Parallel Algorithms and Applications*, 1:43–56, 1993.

[31] Xiru Zhang, Michael McKenna, Jill P. Mesirov, and David Waltz. An efficient implementation of the backpropagation algorithm on the Connection Machine CM-2. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 801–809, San Mateo, CA, 1989. Morgan Kaufman.

Biography

Lutz Prechelt works as senior researcher at the School of Informatics, University of Karlsruhe. There he also received his Master's (1990) and his Ph.D. (1995) in Informatics. His research interests include software engineering (in particular using an empirical research approach), compiler construction for parallel machines, constructive neural network learning algorithms, measurement and benchmarking issues, and research methodology. Prechelt is a member of IEEE CS, ACM, and GI.