Data Locality and Load Balancing for Parallel Neural Network Learning

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät Informatik
Universität Karlsruhe
76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/694092

April 9, 1995

Abstract

Compilers for neural network learning algorithms can achieve near-optimal co-locality of data and processes *and* near-optimal balancing of load over processors for irregular problems. This is impossible for general programs, but restricting programs to that particular problem 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.

Compilation techniques and a compiler implementation for the MasPar MP-1 is described and quantitative results for the effects of various optimizations used in the compiler are given. Experiments with weight pruning algorithms yielded speedups of 28% due to load balancing, and of 195% due to data locality. Two other optimizations, connection allocation and selecting the number of replicates, speed programs up by about 50% or 100%, respectively.

Keywords: compiler optimizations, high level language, portable machine-independent parallel programming, 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, and 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.

A reason for this lack is the fact that today it is so difficult to program distributed memory machines. No compilers exist that combine both efficiency and portability of programs. Either the programmer has to define explicitly the distribution of data and processes over the machine, making programming cumbersome and the resulting programs unportable. Or the language defines an abstraction of the machine in order to allow for portable programs, sacrificing efficiency when it comes to programs working on irregular, dynamically changing data structures.

In this work, I present a compiler that translates machine independent 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. The approach assumes a parallel distributed memory machine with hundreds or thousands of processors that is well-balanced with respect to communication versus computation performance, so that a fine-grained implementation can be efficient.

The state of the art of compiling general-purpose parallel programming languages can be summarized as follows:

1. Data distributions with high data locality can be found for regular problems in array-based data parallel languages using static analysis of index expressions.
2. Load balancing can be performed completely dynamically, sacrificing data locality.
3. Data locality and load balancing can be optimized statically for irregular problems using graph partitioning.
4. Graph partitioning could also be used at run time for irregular problems that dynamically change their structure. In most cases, however, this is prohibitively expensive with respect to run time.

The basic idea of my work is that for neural network training algorithms a compiler can find inexpensive data and load distribution schemes that work well even for problems with dynamically changing structure, if it 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.

Two different approaches suggest themselves for how to provide such semantically rich descriptions of 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 used. In my research, I have used the latter approach, not because it is better, but because it is much easier to implement. 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 [10]. For a description of CuPit see [13].

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 slowly.

The results obtained with a prototype CuPit compiler indicate that the approach is useful: Over a set of benchmark problems, program versions with load balancing were found to be 28% faster than unbalanced ones. With the data locality provided by the compiler the programs executed 195% faster than without. Comparison with a good optimizing compiler for a general-purpose data parallel language showed that the code of the CuPit compiler is competitive even for regular problems with static structure.

The following sections will shortly discuss related work and then describe CuPit's view of neural networks and neural algorithms, the approach used to achieve data locality and load balancing, some implementation details of a compiler prototype, and the results obtained with the prototype.

2 Related work

Data locality can be optimized statically for languages with array-based data parallelism, e.g. [3, 12]. Index analysis is used to compute a good data distribution for programs using mostly linear index expressions, which is the most frequent case. However, in the neural algorithm domain no explicit

analysis of data dependencies is needed: Operations either use only local data or have dependencies that are restricted by the current connection topology of the neural network. Data distributions that take these restrictions into account can exhibit high data locality without using sophisticated program analysis.

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 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.

A class of methods trying to solve the data locality and load balancing problem at once is based on graph partitioning; [7] 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 extremely expensive or produce poor results. The long running time of the better methods forbids to use them repeatedly at run time. Graph partitioning is also not readily applicable to neural network implementations based on connection parallelism or to neural algorithms that do not process all nodes in parallel, as is the case for the most popular class of layered networks. The problem is that the methods assume that all considered objects (here: all nodes and connections of the network) are worked on at the same time.

I am not aware of research for optimizing simulations 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 (e.g. [9] and references therein), mapping of more general static neural networks to high-latency parallel machines (e.g. [18] and references therein, [17] is a bibliography), or very coarse-grained approaches on workstation clusters (e.g. [8]).

3 What is a neural network?

Let us define *neural networks* and *neural algorithms* (neural network training algorithms) as suited to our needs. Since we are concerned with compilation, the description uses programming language terminology. Familiarity with common 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*. The data structure consists of fields called *data elements*, just like in 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* that can accept only either incoming or outgoing connections of a single connection type.

A *node* may have arbitrary data elements. In addition, there are a number of interfaces to the node, each defined by an *interface mode* (either "incoming" or "outgoing") and a connection type. Data elements and interfaces together determine a *node type*.

Nodes are aggregated into *node groups*. A node group type specifies the node type of its elements. Objects of a particular node group type can consist of zero or more nodes of that node type.

A *network* may have arbitrary data elements. In addition, a network type declares a fixed number of node groups to be parts of the network. The initial status of a network object is that all node groups consist of zero nodes.

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.

3.2 Neural algorithm

A *neural algorithm* is a program that manipulates a neural network (as described above) with operations of only the following kinds: A sequential program called the *central agent*, which controls the learning algorithm, can read and write data from and to the nodes of the network and can call *network procedures*. Network procedures can (1) manipulate the data elements of the network object, (2) call *node procedures* to be executed for all or some of the nodes of a particular node group that is part of the network, (3) create or delete nodes in a node group, (4) create or delete connections between a particular pair of interfaces of some or all of the nodes of two node groups, and (5) compute a reduction over a particular data element of some or all nodes of a node group using an arbitrary reduction operator.

Node procedures can (1) manipulate the data elements of the node object, (2) call *connection procedures* to be executed for all of the connections attached to a particular interface of the node, (3) delete the node they are applied to or create multiple copies of the node (including cloning of all the connections), and (4) compute a reduction over a particular data element of all connections attached to a particular interface using an arbitrary reduction operator.

Connection procedures can (1) manipulate the data elements of the connection object, and (2) delete the connection object they are applied to.

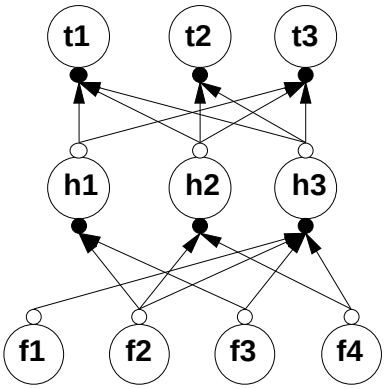
Note that calls to node procedures and connection procedures imply parallelism. Network procedures, node procedures, and connection procedures operate only on the local data elements of the particular network, node, or connection object they are applied to and on the parameters that are supplied with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

In addition, we define *network replication* to mean the following: Creating network replicates means to make identical copies of a network; merging network replicates means to unify the data in all network in a set of replicates by means of elementwise reduction operations (defined by a user program) and redistribution of the results; deleting network replicates means to create a single network from a set of replicates by merging without redistribution; executing a network operation on replicates 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.

Network replication essentially implies parallelism on the level of training examples in a neural algorithm. Note that this kind of parallelism is not applicable to all training algorithms.

The above formulation of neural algorithms gives us three 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; a node procedure invokes a connection procedure on several connections in parallel.

3.3 Example



A 3-layer feed forward neural network with irregular connection topology.

Figure 1: Example neural network

As an example, consider the neural network shown in figure 1. It has three node groups f, h, and t, containing 4, 3, and 3 nodes, respectively. The types of the nodes within each group are the same but the types of, say, $f1$ and $h1$ may differ. There is only one connection type. Each node has two interfaces, one for incoming connections and one for outgoing connections. Connections exist only between nodes of adjacent node groups — the network is called *layered*. f is called an input layer, because it has no incoming connections, t is called an output layer, because it has no outgoing nodes, h is called a hidden layer, because it is neither an input nor an output layer. The data elements of the network, node, and connection types depend on the neural algorithm that is to be implemented. Typical backpropagation-type algorithms will at least have in the connections: elements for weight, delta, input, and output value; in the nodes: elements for input value, output value, and accumulated error; and in the network itself: an element for accumulated error.

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. This means that the majority of 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 neighborhood relations in the communication network of the parallel machine. (3) Broadcasts can be replaced by replication of data and 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, since a node should be local with all its connections, which can be achieved by replicating it several times, yet will often have different numbers of connections at different interfaces.

The approach described below is suitable for implementation on any distributed memory parallel machine, SIMD or MIMD, with explicit distribution and movement of data. Each processor is assumed to have its own local memory. For machines that perform extensive cacheing or implicit data distribution (such as the KSR), the considerations become more complicated, although most of the techniques and analyses described still apply. Synchronization issues will be ignored; they are not critical in neural algorithms. In the following, we will assume a machine with a static interconnection network in which *neighborhoods* can easily be identified, i.e., the machine can be segmented into parts that all have a significantly smaller diameter than the machine as a whole. All of the common interconnection networks such as meshes, trees, and hypercubes have this property. Without it, reductions over connections become less efficient but all other techniques are still applicable. For simplicity of description, we will assume a 2-D grid as the interconnection network for the rest of the text; other topologies can be handled analogously.

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

A1: To use training example parallelism, partition the machine into 2-rectangular *segments* of several processors each and use one such segment for each network replicate. 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. For other topologies than grids, appropriate analogous definitions of 2-rectangular segments can be found. All segments contain exactly the same data structure, but 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: To use node parallelism, do the following 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 figure 2 for an example of segment and node block partitioning. How the partitioning is actually computed will be described in A7 below.

A3: To use connection parallelism, do the following for each interface of each node: Distribute the connections of the interface over the node's block and virtualize connection operations as needed. See figure 3 for an example of connection distribution over node blocks.

A4: 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 node block. This data replication costs one machine-wide broadcast per network procedure call, a broadcast of the result after each reduction, and some memory. The corresponding broadcast savings are at least equivalent of the above broadcast costs. More savings result if the user program computes additional parameters locally in a node.

A5: Do not replicate connection data at both ends of a connection. 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. Data replication would mean that each change in any data element of a connection had to be send to the opposite end of the connection, whereas with the above architecture, less than half of this traffic is needed if the correct decision is made at which end to put the actual data.

A6: To make reductions of connection data into nodes cheap, choose node blocks to be small-diameter sets of processors. 2-rectangular blocks have almost minimum diameter. Small diameter also speeds up data replication in a node block which is necessary after a reduction. For instance in figure 2, the block $h3$ is sized 2x2 instead of 4x1.

A7: To get load balancing, make the size of each node's block proportional to the work performed on the connections attached to the node. A simple measure of work is the number of connections, a more elaborate measure can be obtained by actually measuring the work at run time. When different interfaces of a node have a different number of connections, node block size can be proportional to the amount of work only on the average. That node blocks are 2-rectangular has two further consequences: First, no exact proportionality between node block size and work can be guaranteed. Node blocks can be too large or small by up to factor 2. Second, not all nodes that have the same amount of work should be handled the same in order to avoid poor processor utilization. The algorithm given below can be used to compute the node block sizes $b_1 \ldots b_k$ of k nodes having connection work equivalents of $w_1 \ldots w_k$ for a segment of S processors; compared to the real algorithm the following one is simplified in that node blocks must be forced to have at least size 1; let $p(n)$ mean 2^n:

$W := \sum_{i=1}^{k} w_i$;
Forall $i \in [1 \ldots k]$:
  $s_i := S \cdot w_i / W$; (should-be 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 find the maximum $\alpha \in [1 \ldots 2]$ so that
  $\sum_{i, r_i < \alpha} u_i + \sum_{j, r_j \geq \alpha} d_j \leq S$
now let J be the sequence of relevant 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
  $b_i := u_i, \quad b_c := u_c, \quad b_j := d_j$

From this set of node block sizes the actual node block layout is computed by a bin packing algorithm. The special 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 is shown and discussed in figures 2 and 3 in the next section.

A8: No optimization of extra-object locality is performed 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. [18]). With only small gains in locality, the high running time of the optimization computation takes too long to amortize. Second, arrangement of nodes for optimal extra-object locality interferes with arrangement of nodes 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 prohibitively expensive.

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 now superseded by a faster model, but is a good basis for this work because of two properties: First, due to the large number of processors scaling can be explored

properly. Second, the machine's communication performance is well balanced with its computation performance. To communicate an arbitrary permutation of 4-byte packets over all processors using the general *global router* communication network takes about as long as 30 single precision floating point multiplications. Fetching takes the same amount of time as sending. Sending or fetching 4-byte packets to or from the nearest neighbors using the specialized *xnet* grid communication network takes only 3 multiplies, communication with neighbors 20 processors away in the same direction for all participating processors takes as long as 6 multiplies. For 32-byte packets, the respective values are roughly 100, 10, and 30, respectively. Since neural algorithms need a mix of these communication modes, we find a cost on the order of 5 to 20 floating point multiplications for each floating point communication.

The CuPit compiler was implemented using the Eli compiler construction system [6] and generates MPL code, MasPar's data parallel variant of C. The source code of the compiler is available as a literate programming document [15]. Some details of the implementation and a data distribution example are given in the following sections.

5.1 Networks

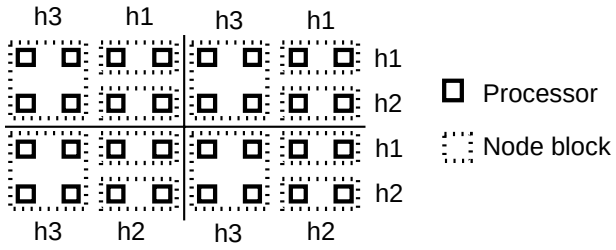
Network replicates are created only on request from a user program. While network replicates exist, no changes in network topology are allowed in CuPit. After topology changes, the network data structure is completely reorganized when replicates are created again or upon program request. Calls to network procedures are executed on all processors of the segment. Virtualization of network procedures is never necessary, since having more replicates than processors does not make any sense.

CuPit allows the programmer to specify the number of network replicates as an interval; the program may choose any number of replicates from this interval at run time. The compiler should generate code to select that number of replicates which leads to fastest execution. However, to determine this optimal number of replicates is quite difficult: The best value 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 — and the one used in the compiler — is to generate code that makes the program iteratively search for optimal values at run time, using changes in run time per training iteration as a hint for when to restart 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 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 the data replication invariant. Virtualization is done for node procedures as needed.

Figure 2 shows an example layout of node blocks. What you see is the node group of layer h of the network from figure 1; the network is replicated fourfold. In the example network, the work required for the connections is about the same for h_1 and h_2, while h_3 requires about twice as much, so the layout given above results (see also figure 3). Since the $h3$ block is the largest, it was placed first and is therefore to the left of the $h1$ and $h2$ blocks.



Example node block layout of layer h of the example network on a 4x8 processor grid using 4 replicates. Replicate boundaries are indicated by straight lines.

Figure 2: Segments and node blocks

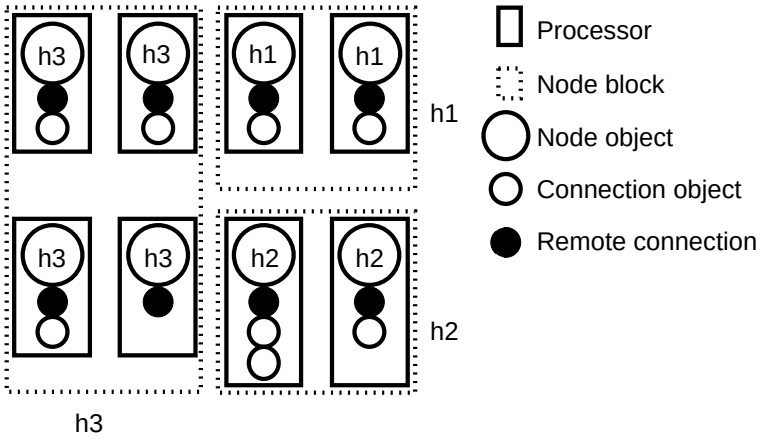
5.3 Connections

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. This decision should not be based on static program analysis, because the optimal data allocation will usually be data dependent. It could be made based on run time analysis of the program, but currently no automatic scheme is implemented in the compiler — connection allocation is changed via compiler options 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. Code for each connection procedure is generated in three parts: One procedure L for the operation on a single connection object as specified in the user program; one procedure VL for virtualization over local connection objects (using L); and one procedure RL for virtualization over remote connection objects (also using L). RL makes L believe it is working on a local connection object by constructing such a local object before L is called: The relevant elements of the remote connection are fetched and a local connection object is initialized with these values. Then L is executed on the local object and the elements that have changed are sent back to the remote connection object.

The CuPit compiler computes the sets of elements to fetch and send 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 pretty well for typical neural algorithms for normal user program coding style, but could 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.

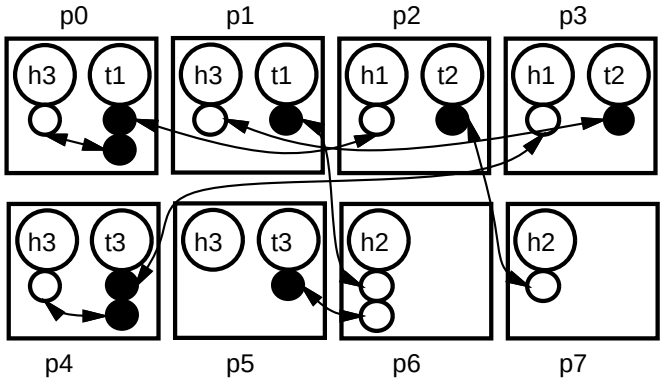
Since node blocks are 2-rectangular, reductions over a data element of the connections of a node use the efficient xnet neighbor communication network on the MasPar. Figure 3 shows the data distribution inside the node blocks of figure 2 for one segment. Each processor contains one node object copy, 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 approximate proportion to the amount of connection work to be done by each node. 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 4 shows the distribution of connections and corresponding remote connections between layers h and t of the example network from figure 1 for one segment. The distribution shown in the figure uses



Distribution of node, connection, and remote connection objects of layer h of the example network from figure 1 within one segment from figure 2.

Figure 3: Node blocks and connections



Layout of nodes of layers 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 4: Connections and remote connections

the straightforward node block sizes; in contrast to these, the node block size computation algorithm given above would assign 4 processors to $t1$ in order to avoid wasting processors. Extrapolating the situation shown in the figure to more and larger layers on more processors and more connections between them illustrates why it is hard to arrange nodes so that connections and remote connections are on the same processor significantly more often than by random assignment.

5.4 Miscellaneous details

All objects carry along descriptors that indicate their validity (existence) and other data as needed: Networks know their replicate number and segment size. Node groups know their number of nodes and factor of virtualization. Nodes know their index and their block size. Connection interfaces know their factor of virtualization, 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; creation of nodes is done by 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.

6 Results and discussion

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

1. To measure the improvements made through 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.
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 general purpose high level language compiler 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.

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 can be decided in different ways, for instance based on weight (idea: small weights are probably not too important) or based on a statistical measure of significance of being non-zero. The latter method was used in the experiments. See [4] for a detailed description.

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. While such a static pruning schedule is not the way pruning would be used on real-life learning tasks, it is sufficiently close to real pruning schedules for our timing measurement purposes.

As experiments showed, the actual pruning criterion is 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 can be gained by load balancing.

For the experiments reported here, 11 real training problems from 10 different domains were used, all taken from the PROBEN1 benchmark set [14]. The name and size of each problem is given in the first four columns of figure 5. The actual datasets for all these problems contain twice as many training examples as indicated in the table. Each problem exists in three different variations created by selecting three different subsets of training examples at random.

For all these 33 problem variations, 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 (called *dbal* for "dumb balancing"), and one without load balancing (called *nbal*, for "no balancing"). The timings reported

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

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 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.

Figure 5: Problem sizes and relative run time of non-optimized program versions

here are based on the time needed for training in epoch 210, i.e., after the last pruning step. The value of *bal* is always used as the basis, normalized to 100.

The results appear in figure 5 (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 128%. Note that the latter value is a conservative estimation of the effect of load balancing for the following reasons: (1) All runs used high numbers of replicates, hampering load balancing capabilities due to small segment sizes (node block sizes cannot differ much). (2) 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. (3) The irregularities in the networks of the example runs were only moderate. Thus, the potential for speedups from load balancing was only moderate, too. (4) Load misbalance of less than factor 2 is sometimes not corrected by the CuPit compiler, since node blocks are always 2-rectangular. Thus, we can expect the actual effects of load balancing to be higher than the 28% mentioned above for most situations.

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 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 or simulate zero latency, performance gains due to load balancing will be significant.

Further experiments (performed using the the Chaco [7] program) showed that using graph partitioning methods to compute the data distributions would not pay off: After pruning two thirds of all connections they could increase remote connection locality only by about 10%, which is not enough to amortize the significant run time they consume on each data redistribution (even after only minor

pruning when savings are only much smaller).

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. The alternative chosen here is similar to what would result from formulating the program in a language with array-based parallelism where arrays are distributed over all processors in some regular fashion. 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. An additional cost of the array-based scheme is that 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 compiler was instrumented to generate code that simulated no connection object data locality for connection operations (but still avoided parameter broadcast). Timing measurements with these non-data-local variants of the otherwise unchanged program produced the results indicated in the *noloc* column of figure 5. 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. Thus, implementations without data locality take about two to three times as long to execute.

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. Doing without data locality or without load balancing might allow for simpler and faster data distribution procedures. I thus measured how much time was spent in the general data distribution procedures. The times were measured in the same experimental setting as described in the section on load balancing. It was found that, 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* program. The latter was translated by a compiler that also targets the MasPar and that is known to generate efficient code [11]. The problem chosen was backpropagation using the RPROP learning rule [16] 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) allowed the Modula-2* compiler to 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: It avoids copying of unnecessary data upon redistribution of the network data after a network replicate merge operation and it fetches and sends only those elements of a remote connection that are really used at run time while CuPit code fetches all elements that *may* be used as determined by a very simple static analysis. Avoiding unnecessary data redistributions after network merge is an optimization that is not implemented in the current CuPit compiler; in the Modula-2* program, redistribution has to be coded by hand.

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 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.

The results indicate that Modula-2* code is 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. Over all examples, the average relative run time of the Modula-2* program was 142%. When the ability of the CuPit compiler to combine multiple fetch or send operations was switched off, this average dropped to 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 comparisons with other compilers were made for irregular problems, since 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

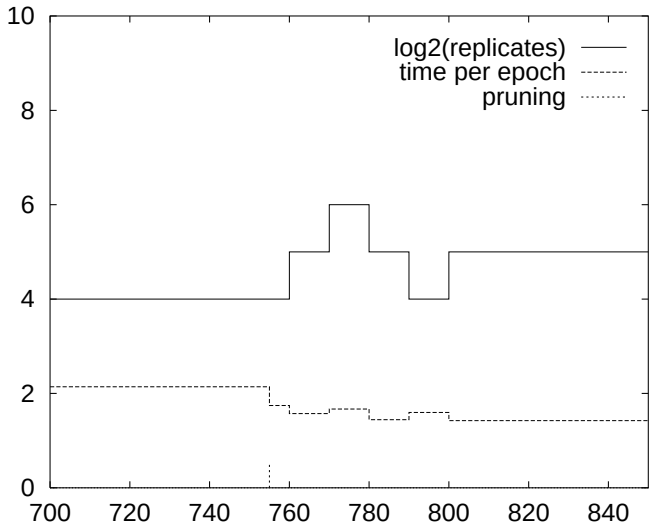
wrong decision in this respect. The experimental setup was exactly as described in the load balancing section; a program called *cwrong* that placed connection objects at output interfaces (which is the wrong decision for this program) was timed. The results are shown in the respective column of figure 5.

As we see there is a significant performance penalty of about 50 percent run time increase for choosing the wrong 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 5, not having communication aggregation (see section 5.3) costs an additional 10% run time on the average for the MasPar. The value would be higher for machines with higher latency to bandwidth ratio.

6.7 Selecting the number of replicates



Changes in number of replicates and corresponding run time per epoch initiated automatically after the pruning step in epoch 755.

Figure 6: Automatic replicate number optimization

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 6 shows an example of how the adaptive search works. The relative run time of the latter is shown in table 5 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.

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.

Over a variety of irregular problems, a prototype implementation of the approach produced the following speedups: 28% due to load balancing, 195% due to data locality, and 54% due to optimal remote connection object placement. The corresponding data distribution computations took 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 produce efficient code for irregular problems from a high-level description automatically. Therefore, the general principles of the approach should also be applied to machines with smaller numbers of processors and be compared with other techniques.

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] 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.

[7] Bruce Hendrickson and Robert Leland. The Chaco user's guide, version 1.0. UC-405 SAND93-2339, Sandia National Laboratories, Albuquerque, NM 87185, October 1993.

[8] 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.

[9] 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.

[10] 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].

[11] 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.

[12] Michael Philippsen and Markus U. Mock. Data and process alignment in Modula-2*. In *AP '93, Int. Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution, and Automatic Parallel Performance Prediction*, pages 141–149, Saarbrücken, Germany, March 1993.

[13] 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/1994/1994-4.ps.Z on ftp.ira.uka.de.

[14] 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.Z on ftp.ira.uka.de.

[15] Lutz Prechelt. The CuPit compiler for the MasPar — a literate programming document. Technical Report 1/95, Fakultät für Informatik, Universität Karlsruhe, Germany, January 1995. Anonymous FTP: /pub/papers/techreports/1995/1995-1.ps.Z on ftp.ira.uka.de.

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

[17] Tom Tollenaere. Neural network simulations on transputers. Technical Report TR 91 0021, Version 1, Katholieke Universiteit te Leuven, Leuven, Belgium, May 1991.

[18] 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.