A Parallel Programming Model for Irregular Dynamic Neural Networks

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

Abstract

The compilation of high-level programming languages for parallel machines faces two challenges: maximizing data/process locality and balancing load. No solutions for the general case are known that solve both problems at once.

The present paper describes a programming model that allows to solve both problems for the special case of neural network learning algorithms, even for irregular networks with dynamically changing topology (constructive neural algorithms). The model is based on the observation that such algorithms predominantly execute local operations (on nodes and connections of the network), reductions, and broadcasts.

The model is concretized in an object-centered procedural language called CuPit. The specific properties of the model are introduced via

1. *special categories (analog to categories "record" or "array" in other languages) of object types: "connection", "node", and "network",*
2. *3-fold nested parallelism, described via group procedure calls (levels: network replicates, node groups, connections at a node), and*
3. *special operations for manipulation of the neural network topology.*

The language is completely abstract: No aspects of the parallel implementation such as number of processors, data distribution, process distribution, execution model etc. are visible in user programs. The compiler can derive most information relevant for the generation of efficient code from unannotated source code. Therefore, CuPit programs are efficiently portable.

A compiler for CuPit has been built for the MasPar MP-1/MP-2 using compilation techniques that can also be applied to most other parallel machines. The paper shortly presents the main ideas of the techniques used and results obtained by the various optimizations.

Key words: Data and process locality, load balancing, compiler, portability.

1 Ease vs. Efficiency

The two most important issues of parallel programming languages are (1) efficiency of implementation and (2) ease of programming. Unfortunately, however, these are usually contradictory. Any attempt to improve the ease of programming usually tends to decrease the efficiency that can be achieved by compilers for the language.

This is mostly a problem of available knowledge: Efficient implementation requires that the compiler has thorough knowledge of the semantics of the program, whereas one important aspect of achieving ease of programming is to free the programmer from having to supply detailed explicit knowledge about program meaning and execution.

Therefore, one path towards useful parallel programming languages is to identify application domains in which much knowledge can easily be extracted from a domain-oriented program description and to define domain-dependent languages for these domains. The two kinds of information that are of particular interest are (1) the dynamic distribution of work over parallel threads (needed for load balancing) and (2) the dynamic distribution of data versus threads (needed to obtain co-locality of data and process).

Neural network learning algorithms, even those that dynamically change the topology of the neural network, are such a domain; we will simply call these class of programs *neural algorithms*. This paper describes how to exploit their special properties in order to obtain a purely problem-dependent (thus easy-to-use) programming model that can nevertheless be implemented efficiently.

The subsequent sections list these special properties, derive a programming model, describe the concretization of this model in a programming language, and give some results obtained with an implementation of this language.

2 Properties of Neural Algorithms

A neural algorithm is a program that performs parallel computations on a graph of nodes and connections (the neural network). For such programs, the following assumptions hold:

A1: The outer loops are structured along the lines of: Read a training example and perform some computations A on it (using all or most of the network elements); after a number of examples are so processed, perform some computation B and perhaps some change C in graph structure (network topology).

A2: There are basically five types of operations: local operations on nodes or connections, reductions, broadcasts (multicasts), and generation and destruction of nodes or connections.

A3: No expressions involving arbitrary pairs of operands occur. Instead, computation is always attached to the objects of the neural network graph in one of the above styles.

A4: Therefore, there is also no arbitrary use of parallelism. There is only parallelism over the connections of a node, over the nodes of one or several node groups, and (for those algorithms that allow for example parallelism) over multiple replicates of a network. These three levels of parallelism are nested.

A5: The computations are homogeneous in the sense that any single parallel operation (procedure call) takes the same time on all of the objects affected, in particular on the innermost (i.e., connection) level.

A6: Co-locality of the data for several nodes that are connected with each other can hardly be improved over that resulting from a random distribution of data over the processors, because the graphs of neural networks do not exhibit much locality. (This does not always apply to *structured* NNs, which we do not consider here.)

3 The Programming Model Idea

The basic idea of the programming model proposed here is to explicitly model the network objects in the programming language and to restrict the types of operations that can be performed on these objects to the operations needed by neural algorithms as given above. To support constructive neural algorithms, special operations are included for the creation and deletion of network, node, and connection objects.

This approach makes a lot of information readily available to the compiler that would be very difficult or impossible to extract from an equivalent program text in a normal parallel programming language such as HPF [4]. The information can then be used to generate efficient code that exhibits almost optimal data/process-locality and balanced load, even for irregular networks. In many respects, the ease with which information about data access patterns can be extracted from the program in this approach is comparable to that found in functional or single assignment languages such as SISAL [1], because in both cases no arbitrary interactions between global data objects are possible. Yet the optimization capabilities arising from this information are still better in our case, since the types of operations are restricted and thus known in advance — allowing to design optimizations for their implementation into the compiler.

4 CuPit

The programming language CuPit [5] is a realization of the programming model described above. The most important features of its design will be described below, mostly using example program fragments instead of formal definitions.

CuPit is a procedural, object-centered language, i.e., there are object types and associated operations but no inheritance. The identification of network elements is based on three special *categories* of object types: There are connection types, node and node group types, and network types.

```
TYPE Weight IS CONNECTION
  Real     i       := 0.0,
           o       := 0.0,
           weight  := 0.0,
           delta   := 0.0;
  PROCEDURE prune (Real CONST pruneThreshold) IS
    IF ME.i <= pruneThreshold
    THEN REPLICATE ME INTO 0;  END;
  END PROCEDURE;

  PROCEDURE transport (Real CONST val) IS
    ME.i := val;
    ME.o := val*ME.weight;
  END PROCEDURE;
END TYPE;
```

The above declaration defines a connection type `Weight`. Objects of this type are a structure of four data elements `i, o, weight,` and `delta`, all of the built-in type `Real`. Associated with this type are two procedures. `prune` implements a conditional self-deletion of the connection (the same could be done for nodes and then includes all their connections) and `transport` implements the forward pass operation through the connection, i.e. store in `i` the input and make the output `o` be the weighted input. `ME` always designates the object, for which an object procedure is being called. Both procedures can only be called from

nodes that have connections of type **Weight** attached; we will see a call in the following example.

```
TYPE SigmoidNode IS NODE
  IN  Weight in;
  OUT Weight out;
  Real       inData;
  Real       outData;
  PROCEDURE forward (Bool CONST doIn, doOut) IS
    IF doIn  THEN
      REDUCTION ME.in[].o:rsum INTO ME.inData;
    END;
    IF doOut THEN
      ME.outData := activation (ME.inData);
      ME.out[].transport(ME.outData);
    END;
  END PROCEDURE;
END TYPE;
```

This node type, **SigmoidNode**, has two data elements, **inData** and **outData**, and two *connection interfaces*, **in** (for incoming connections) and **out** (for outgoing connections), both for connections of the above type **Weight**. The direction of a connection does not constrain how it can be used. How to create connections is shown below. The node procedure **forward** can operate on all connections attached to one of these interfaces at once. For instance the **REDUCTION** statement reduces the **o** elements of all connections attached to the **in** interface using the reduction operator **rsum** shown below. The result is written into the variable **ME.inData**, which will be unchanged if there are no connections. The **[]** notation stands for '*all*' and designates parallel calls.

The reduction operator **rsum** used above is defined by the user as

```
Real REDUCTION rsum IS
  RETURN (ME + YOU);
END REDUCTION;
```

Arbitrary reduction operators on arbitrary data types can be defined this way.

The **activation** procedure called above is a so-called *free procedure*, i.e., one that is not attached to any object type and can be called from anywhere; it returns a **Real** in this case. The call to **transport** (a procedure from the connection type) broadcasts the value **ME.outData** to all connections attached to the interface **out** of the node executing the **forward**. The **transport** procedure is then executed in parallel asynchronously for each of these connections. This call realizes nested parallelism, as **forward** itself is already being executed for several nodes in parallel as we will see below.

```
TYPE Layer IS GROUP OF SigmoidNode END;
```

```
TYPE Mlp IS NETWORK
  Layer   inL, hidL, outL;
  PROCEDURE createNet (Int CONST inputs,
                       hidden, outputs) IS
    EXTEND ME.inL BY inputs;
    EXTEND ME.hidL BY hidden;
    EXTEND ME.outL BY outputs;
    CONNECT ME.inL[0].out  TO ME.outL[].in;
    CONNECT ME.inL[].out TO ME.hidL[].in;
    CONNECT ME.hidL[].out TO ME.outL[].in;
    (* ...initialize weights etc. *)
  END;

  PROCEDURE example () IS
    ME.inL[].forward   (false, true)
    ME.hidL[].forward  (true,  true);
    ME.outL[].forward  (true,  false);
    ME.outL[].backward (false, true);
    ME.hidL[].backward (true,  true);
  END PROCEDURE;
END TYPE;
```

The network type **Mlp** is a simple three layer perceptron, consisting solely of the three node groups **inL, hidL, outL**. A node group is a dynamic, ordered set of nodes. Groups of groups may be useful as well for some programs, but are not currently supported in CuPit. The **createNet** procedure dynamically creates the nodes in the groups and the connections between them. **ME.inL[0]** is a bias node and thus has connections to all nodes of **outL**, while the other nodes of **inL** have not. The **example** procedure executes the forward and backward pass through the network for one input/output example pair. The individual data values for the example are written into the network by the main program before the procedure is called.

A small fraction of a main program is shown in the following fragment:

```
Real  IO   x1, x2; (* extern-managed I/O-areas *)
Mlp   VAR  net;    (* Her majesty, the NETWORK *)

PROCEDURE program () IS
  (* ... *)
  net[].createNet (inputs, hidden, outputs);
  REPLICATE net INTO 1...300;
  REPEAT
    getExamples (x1, x2, REPLICATES(net));
    net.inL[].inData   <-- x1;
    net.outL[].outData <-- x2;
    net[].example ();
    (* ... merging, weight update, etc. *)
  UNTIL stopTraining ()  END REPEAT;
END PROCEDURE;
```

I/O-areas such as **x1** and **x2** above are special sorts of variables with a defined memory layout. This is in

contrast to the actual network data, the memory layout of which is not known to the programmer in order to allow for arbitrary optimizations. I/O areas are accessed by a procedure (here: `getExamples`) written in some native language of the specific parallel machine to move data in and out of the CuPit program using native memory or input/output operations. The operators `-->` and `<--` transfer data between I/O areas and nodes. Subsets of node groups can be accessed using a slice notation, e.g. `net.outL[2...5].inData --> x2` would output the `inData` value from nodes 2 to 5 of `outL` into `x2`. The same notation can be used in `CONNECT` statements.

The statement `REPLICATE net INTO 1...300` requests network replicates to be generated in order to exploit parallelism over examples. The compiler or run time system can chose how many replicates will fit into memory and will execute fastest; any number in the range 1...300 is allowed in this program. The `getExamples` procedure has to supply an appropriate number `REPLICATES(net)` of examples in each call. During training, the replicates will diverge in their data values, but not in their network topology, since topology modifications are forbidden while a network is replicated. To synchronize data in replicates, the program calls `MERGE net` (not shown above), which executes type-specific user-defined data merge operations in all objects. In the above program, merging is required for the `delta` values in the connections only (always just before the weight update step). This merging is thus realized by including the definition

```
MERGE IS
  ME.delta += YOU.delta;
END MERGE;
```

in the type `Weight`. All other administration of network replicates is completely implicit. To reunite replicates into just one network (for instance in order to perform topology changes on the network) one would call `REPLICATE net INTO 1`, which also performs a merge first.

Topology modification statements not shown in the above program are `DISCONNECT` (inverse to `CONNECT`), node cloning (e.g. `REPLICATE ME INTO 3`, which triplicates the node and all its connections), and node removal using negative arguments to `EXTEND`.

This language allows for the convenient modular specification of constructive neural learning algorithms, no matter whether these add resources during learning or remove resources or both. Algorithms such as the additive CasCor method [3] or the Optimal Brain Damage pruning method [2] can be expressed easily. CuPit programs are explicitly parallel for the compiler, yet natural for the programmer, because the domain metaphor of the neural network is used to express the parallelism. Most information needed for program optimization can readily be extracted from the program text by a compiler.

5 Alternative Realization: o-o languages

Instead of realizing the programming model in a special purpose language such as CuPit, the same ideas could also be embedded in parallel object-oriented languages. To do this, several base classes have to be defined: *connection, interface, node, node group*, and *network*. Each user-defined, say, connection type inherits from the *connection* base type, etc.

To fully exploit the information available in the programming model, such an implementation should build some knowledge about the behavior of the program (with respect to the special object types) into the compiler. Parts of this knowledge have to be enforced by restrictions on the use of the types, which should also be checked by the compiler. For instance it must be forbidden to create a connection object without attaching it to exactly one input and one output node interface.

The obvious advantages of this approach are that the programmer does not have to learn a complete new language and that the integration with other parts of an application becomes simpler. The disadvantages, on the other hand, are that the approach is technically more difficult to realize and that artificial constraints have to be imposed on the programming style, which may result in actually more difficult programming than with a (albeit newly learned) special purpose language.

6 Implementation

A prototype CuPit compiler has been built for the MasPar MP-1/MP-2 massively parallel SIMD machine. The compiler source code is available as a literate programming document [8].

Put very shortly, the following techniques and optimizations are employed in the compiler:

O_1: Data locality is maintained by locating node objects and the associated connection objects on the same processor. Since each connection is attached to two nodes, this is possible only in a little more than half of all cases. The node at the opposite end has only a pointer to the connection object (remote connection).

O_2: Data locality is maximized by chosing the 'right' end of the connections (input or output end) as the pointer end, namely the one which is used less often or with less data during the program run.

Problem	N_{in}	N_{out}	N_{ex}	O_1	O_2	O_3	O_4	O_5
building	14	3	2104	244	145	123	110	99
flare	24	3	533	289	141	120	112	165
hearta	35	1	460	325	151	114	111	110
cancer	9	2	350	305	144	150	97	98
card	51	2	345	333	164	139	108	120
diabetes	8	2	384	294	159	129	110	161
gene	120	3	1588	221	149	115	119	77
glass	9	6	107	309	165	130	101	102
heart	35	2	460	320	153	130	111	110
soybean	82	19	342	288	167	132	115	130
thyroid	21	3	3600	247	144	120	121	114
(average)	35.8	3.8	871	289	154	127	110	115

N_{in}, N_{out}, N_{ex}: Number of input nodes, output nodes, and training examples, respectively. O_x: relative run times (in percent) of pruning program with optimization O_x switched off versus switched on.

Table 1: Problem sizes and relative run time of non-optimized program versions

O_3: Load balancing is ensured by allocating for each node and its connections an amount of processor resources that is proportional to the work that needs to be done on average per connection. The weighting factors for the input versus output connections are determined by direct measurement during program runs. The administrative overhead of measurement and the actual data distribution procedure is much smaller than the gains.

O_4: Accesses to data elements from remote connections are bundled over a connection operation. Due to the language structure, a simple static textual analysis is sufficient to determine almost optimal sets of elements for each operation.

O_5: The number of network replicates used is adapted dynamically during run time by a direct search method using halving/doubling steps.

7 Results

The compiler and optimizations was evaluated on a variety of neural network learning problems, which are described in [6] Table 1 presents the relative changes in run time for each of these problems on a 16384 processor MasPar MP-1, when each one of the optimizations in turn was switched off.

All experiments used the lprune pruning algorithm [7], because the size and topology of the neural network has a much larger impact on the results than the choice of the actual learning program.

For the load balancing (O_3) results it must be noted that the values were measured after only two thirds of the connections had been removed. Thus, the network in these measurements had only very moderate irregularity. More irregular networks will result in larger gains from load balancing.

The basis of comparison for the dynamic selection of replicate numbers (O_5) was a static number of replicates, which was plausible (in fact, it was my personal educated guess) but not in all cases optimal. The bad result on the gene problem is due to the simple-minded implementation of the search method.

As we see, the various optimizations result in moderate to dramatic performance gains, depending significantly on the actual problem to be solved. These optimizations could hardly be realized without the ability to easily extract the required information from the CuPit program source text.

8 Conclusion

A domain-dependent programming model and programming language (CuPit) for constructive neural algorithms was presented. Compilers can easily extract knowledge about program behavior from such programs. This knowledge can be used for the generation of efficient code, in particular to maximize data/process locality and load balancing, which is otherwise very difficult for problems with an irregular and dynamically changing structure as they arise for instance from constructive neural algorithms.

References

[1] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[2] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *[9]*, pages 598–605, 1990.

[3] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation learning architecture. In *[9]*, pages 524–532, 1990.

[4] High Performance Fortran (HPF): Language specification. Technical report, Center for Research on Parallel Computation, Rice University, 1992.

[5] 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.

[6] 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.

[7] Lutz Prechelt. Adaptive parameter pruning in neural networks. Technical Report 95-009, International Computer Science Institute, Berkeley, CA, March 1995.

[8] 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.

[9] David S. Touretzky, editor. *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. Morgan Kaufman Publishers Inc.